Design and Implementation of WCET Analyses

Including a Case Study on Multi-Core Processors with Shared Buses

A dissertation submitted
towards the degree Doctor of Engineering
of the Faculty of Mathematics and Computer Science
of Saarland University

by

Michael Jacobs

Saarbrücken, 2021

UNIVERSITÄT
DES
SAARLANDES

Day of Colloquium	2021-09-10
Dean of the Faculty	Univ.-Prof. Dr. Thomas Schuster
Chair of the Committee	Prof. Dr. Holger Hermanns
First Reviewer	Prof. Dr. Sebastian Hack
Second Reviewer	Prof. Dr. Reinhard Wilhelm
Third Reviewer	Prof. Dr. Christine Rochange
Academic Assistant	Dr. Valentin Touzeau

Abstract

For safety-critical real-time embedded systems, the *worst-case execution time* (WCET) analysis—determining an upper bound on the possible execution times of a program—is an important part of the system verification. Multi-core processors share resources (e.g. buses and caches) between multiple processor cores and, thus, complicate the WCET analysis as the execution times of a program executed on one processor core significantly depend on the programs executed in parallel on the concurrent cores. We refer to this phenomenon as *shared-resource interference.*

This thesis proposes a novel way of modeling shared-resource interference during WCET analysis. It enables an efficient analysis—as it only considers one processor core at a time—and it is sound for hardware platforms exhibiting timing anomalies. Moreover, this thesis demonstrates how to realize a timing-compositional verification on top of the proposed modeling scheme. In this way, this thesis closes the gap between modern hardware platforms, which exhibit timing anomalies, and existing schedulability analyses, which rely on timing compositionality. In addition, this thesis proposes a novel method for calculating an upper bound on the amount of interference that a given processor core can generate in any time interval of at most a given length. Our experiments demonstrate that the novel method is more precise than existing methods.

Zusammenfassung

Die Analyse der maximalen Ausführungszeit (*Worst-Case-Execution-Time*-Analyse, WCET-Analyse) ist für eingebettete Echtzeit-Computer-Systeme in sicherheitskritischen Anwendungsbereichen unerlässlich. Mehrkernprozessoren erschweren die WCET-Analyse, da einige ihrer Hardware-Komponenten von mehreren Prozessorkernen gemeinsam genutzt werden und die Ausführungszeit eines Programmes somit vom Verhalten mehrerer Kerne abhängt. Wir bezeichnen dies als *Interferenz durch gemeinsam genutzte Komponenten*.

Die vorliegende Arbeit schlägt eine neuartige Modellierung dieser Interferenz während der WCET-Analyse vor. Der vorgestellte Ansatz ist effizient und führt auch für Computer-Systeme mit Zeitanomalien zu korrekten Ergebnissen. Darüber hinaus zeigt diese Arbeit, wie ein zeitkompositionales Verfahren auf Basis der vorgestellten Modellierung umgesetzt werden kann. Auf diese Weise schließt diese Arbeit die Lücke zwischen modernen Mikroarchitekturen, die Zeitanomalien aufweisen, und den existierenden Planbarkeitsanalysen, die sich alle auf die Kompositionalität des Zeitverhaltens verlassen. Außerdem stellt die vorliegende Arbeit ein neues Verfahren zur Berechnung einer oberen Schranke der Menge an Interferenz vor, die ein bestimmter Prozessorkern in einem beliebigen Zeitintervall einer gegebenen Länge höchstens erzeugen kann. Unsere Experimente zeigen, dass das vorgestellte Berechnungsverfahren präziser ist als die existierenden Verfahren.

Extended Abstract

Safety-critical real-time embedded systems—as e.g. found in the automotive domain—have to fulfill strict timing requirements. As a consequence, the timing verification is an important part of the overall verification of these systems. The timing verification usually consists of two steps. First, a *worst-case execution time* (WCET) analysis determines an upper bound on the possible execution times per program. Subsequently, a schedulability analysis uses the resulting set of WCET bounds in order to determine whether the set of programs is guaranteed to be schedulable in a way that the timing requirements are fulfilled. This thesis focuses on WCET analysis.

Multi-core processors share resources (e.g. buses and caches) between multiple processor cores in order to reduce the price, weight, and energy consumption of computer systems. To exploit these advantages, multi-core processors are increasingly used in safety-critical embedded systems. However, the resource sharing complicates the WCET analysis as the execution times of a program executed on one processor core significantly depend on the programs executed in parallel on the concurrent cores. We refer to this phenomenon as *shared-resource interference*. A detailed consideration of this interference during WCET analysis is intractable as it would correspond to the enumeration of all interleavings of access requests to the shared resources. Thus, most existing approaches resort to a more modular analysis—at the cost of a reduced precision. Additionally, all of these modular analyses (often only implicitly) rely on the principle of *timing compositionality*. However, it has so far been unclear how to safely apply such timing-compositional analyses to programs executed on hardware platforms exhibiting timing anomalies. Recent results indicate that even simple micro-architectures—previously believed to be anomaly-free—exhibit timing anomalies.

This thesis proposes a novel way of modeling shared-resource interference during WCET analysis. It enables a modular analysis—as it only considers one processor core at a time—and it is sound for hardware platforms exhibiting timing anomalies. Moreover, this thesis demonstrates how to realize a timing-compositional verification on top of the proposed modeling scheme. In this way, this thesis closes the gap between modern hardware platforms, which exhibit timing anomalies, and existing schedulability analyses, which rely on timing compositionality.

Most modular approaches to the timing verification of systems with multi-core processors rely on the calculation of values on arrival curves. In this context, a value on an arrival curve upper-bounds the amount of interference that a processor core can generate in any time interval of at most a given length. Existing methods for calculating values on arrival curves pessimistically assume that the overall amount of interference that an execution run of a considered program generates can be distributed across the execution run in an arbitrary manner. In this thesis, we propose a more fine-grained calculation method that takes into account which part of a program can generate which amount of interference. The implementation of this calculation method is based on a novel variant of implicit path enumeration that argues about all possible subpaths through the considered programs. Our experiments demonstrate that the novel method is on average about five percent more precise than existing methods.

Last but not least, this thesis makes conceptual contributions to the formal derivation of WCET analyses. The novel principle of *property lifting* enables the use of properties of the concrete system under analysis for the detection of infeasible abstract traces at the level of approximation on which WCET analyses operate. We demonstrate the application of property lifting during the formal derivation of our WCET analysis for multi-core processors with shared buses. Moreover, it can be used for the formal derivation of a wide range of existing approaches which use additional constraints in order to prune infeasible abstract traces during the calculation of a WCET bound. These existing approaches were previously derived mostly based on intuition.

Ausführliche Zusammenfassung

Eingebettete Echtzeit-Computer-Systeme für sicherheitskritische Anwendungsbereiche – wie zum Beispiel den Einsatz in Automobilen – müssen bezüglich ihres Zeitverhaltens strenge Anforderungen erfüllen. Folglich ist die Verifizierung des Zeitverhaltens ein wichtiger Teil der Verifizierung dieser Systeme. Die Verifizierung des Zeitverhaltens wird üblicherweise in zwei Einzelschritte unterteilt. Zunächst bestimmt eine Analyse der maximalen Ausführungszeit (*Worst-Case-Execution-Time*-Analyse, WCET-Analyse) pro Programm eine obere Schranke der möglichen Ausführungszeiten. Anschließend nutzt eine Planbarkeitsanalyse die resultierende Menge an Zeitschranken um festzustellen, ob in Zusammenhang mit dem verwendeten Ablaufplanungsalgorithmus in jedem Fall alle Anforderungen hinsichtlich des Zeitverhaltens erfüllt werden. Die vorliegende Arbeit befasst sich hauptsächlich mit der WCET-Analyse.

In Mehrkernprozessoren werden einige Hardware-Komponenten (z.B. Übertragungswege oder Zwischenspeicher) von mehreren Prozessorkernen gemeinsam genutzt um den Preis, das Gewicht und den Stromverbrauch der Prozessoren zu senken. Wegen dieser Vorteile werden Mehrkernprozessoren auch immer häufiger in eingebetteten Echtzeit-Computer-Systemen für sicherheitskritische Anwendungen eingesetzt. Die gemeinsam genutzten Komponenten erschweren jedoch die WCET-Analyse, da durch sie die Ausführungszeit eines auf einem Kern ausgeführten Programmes von den gleichzeitig auf den anderen Kernen ausgeführten Programmen abhängt. Wir bezeichnen dies als *Interferenz durch gemeinsam genutzte Komponenten.* Eine genaue Betrachtung dieser Interferenz während der WCET-Analyse ist gleichbedeutend mit der Aufzählung aller möglichen Reihenfolgen an Zugriffen auf die gemeinsam genutzten Komponenten durch die verschiedenen Prozessorkerne und hat sich als zu aufwändig erwiesen. Daher setzen die meisten existierenden Verfahren zur Verifizierung des Zeitverhaltens von Mehrkernprozessoren auf ein modulareres Vorgehen, das jedoch zu einer verringerten Präzision führt. Zusätzlich verlassen sich all diese modularen Verfahren (oft sogar nur implizit) auf eine *Kompositionalität des Zeitverhaltens.* Bisher ist jedoch nicht bekannt, wie solche zeitkompositionalen Verfahren zur Verifizierung von Computer-Systemen mit Zeitanomalien genutzt werden können ohne dabei möglicherweise die tatsächliche Ausführungszeit eines Programmes zu unterschätzen. Aktuelle Forschungsergebnisse legen nahe, dass sogar relativ einfache Mikroarchitekturen, die in diesem Zusammenhang bisher als unproblematisch galten, Zeitanomalien aufweisen können.

Die vorliegende Arbeit schlägt eine neuartige Modellierung der Interferenz während der WCET-Analyse für Mehrkernprozessoren vor. Der vorgestellte Ansatz erlaubt ein modulares Vorgehen, da jeweils nur das Ausführungsverhalten eines Prozessorkernes gleichzeitig betrachtet wird, und führt auch für Computer-Systeme mit Zeitanomalien zu korrekten Ergebnissen. Darüber hinaus zeigt diese Arbeit, wie ein zeitkompositionales Verfahren auf Basis der vorgestellten Modellierung umgesetzt werden kann. Auf diese Weise schließt diese Arbeit die Lücke zwischen modernen Mikroarchitekturen, die Zeitanomalien aufweisen, und den existierenden Planbarkeitsanalysen, die sich alle auf die Kompositionalität des Zeitverhaltens verlassen.

Die meisten modularen Verfahren zur Verifizierung des Zeitverhaltens von Computer-Systemen mit Mehrkernprozessoren berechnen zudem obere Schranken der möglichen Anzahlen an Zugriffen auf die gemeinsam genutzten Komponenten in Abhängigkeit von der Länge des jeweils betrachteten

Zeitintervalles. Bisher geht die Berechnung solcher Schranken pessimistisch davon aus, dass die Gesamtzahl an Zugriffen einer Programmausführung beliebig über die Programmausführung verteilt werden kann. In der vorliegenden Arbeit stellen wir eine feinkörnigere Berechnung vor, die berücksichtigt, welcher Teil eines Programmes welche Menge an Zugriffen auf die gemeinsam genutzten Komponenten macht. Die vorgestellte Berechnung ist implementiert durch eine neuartige Variante der impliziten Pfadaufzählung, die über alle teilweisen Pfade durch die betrachteten Programme argumentiert. Unsere Experimente zeigen, dass das vorgestellte Berechnungsverfahren im Schnitt um etwa fünf Prozent präziser ist als die existierenden Verfahren.

Außerdem leistet die vorliegende Arbeit einen konzeptionellen Beitrag zur formalen Herleitung von WCET-Analysen. Das neuartige Prinzip des *Property-Lifting* erlaubt die Verwendung von Eigenschaften eines konkreten Computer-Systems zur Identifizierung von nicht tatsächlich möglichen Ausführungsverhalten auf der Abstraktionsebene, auf der WCET-Analysen argumentieren. Wir veranschaulichen die Verwendung des Property-Lifting im Zuge der formalen Herleitung unserer WCET-Analyse für Mehrkernprozessoren mit gemeinsam genutzten Übertragungswegen. Darüber hinaus kann Property-Lifting zur formalen Herleitung einer breiten Palette an existierenden WCET-Analysen verwendet werden, die zusätzliche Ungleichungen verwenden um die Präzision der berechneten Zeitschranken zu erhöhen. Diese existierenden WCET-Analysen wurden bisher nicht auf formale Weise hergeleitet, sondern lediglich basierend auf Beispielen und Intuitionen.

Résumé détaillé

Les systèmes temps-réel critiques – comme ceux utilisés par exemple dans le domaine automobile – doivent satisfaire des exigences strictes au regard de leur réponses temporelles. En conséquence, la vérification de la réponse temporelle est une partie importante de la vérification de ces systèmes. Habituellement, cette vérification se compose de deux étapes. D'abord, une analyse de temps d'exécution dans le pire des cas (analyse WCET, *worst-case execution time*) détermine une borne supérieure pour les temps d'exécution de chaque programme. Ensuite, une analyse d'ordonnancement utilise les bornes WCET pour prouver l'ordonnançabilité de l'ensemble des programmes. Cette thèse de doctorat se concentre sur l'analyse WCET.

Les processeurs multi-cœurs partagent certaines ressources (par exemple des mémoires cache ou des bus) entre plusieurs cœurs pour réduire le prix, le poids et la consommation d'énergie des systèmes. Pour profiter de ces avantages, on incorpore de plus en plus des processeurs multi-cœurs aux systèmes temps-réel critiques. Cependant, le partage de ressources complique l'analyse WCET, car le temps d'exécution d'un programme dépend du comportement de plusieurs cœurs. Ce phénomène est appelé *interférence due au partage de ressources*. Une considération détaillée de cet interférence pendant l'analyse WCET est en pratique trop exigeante, parce qu'elle est équivalente à une énumération exhaustive de toutes les séquences possibles dans lesquelles les cœurs différents peuvent avoir accès aux ressources partagées. Ainsi, la plupart des approches existantes recourent à une analyse plus modulaire – au prix d'une précision réduite. De plus, toutes ces approches modulaires comptent (souvent seulement implicitement) sur la *compositionalité de la réponse temporelle*. Jusqu'à présent, l'emploi de ces approches compositionnelles en temps d'une manière fiable dans les systèmes qui comportent des anomalies temporelles n'était pas évident. Les derniers résultats de recherche indiquent que même des micro-architectures simples – précédemment supposées être sans anomalies – peuvent comporter des anomalies temporelles.

Cette thèse de doctorat propose une nouvelle manière de modeler l'interférence pendant l'analyse WCET. La manière proposée permet une analyse modulaire – parce qu'elle considère seulement un cœur à la fois – et est correcte pour les systèmes qui comportent des anomalies temporelles. De plus, cette thèse démontre comment réaliser une vérification compositionnelle en temps sur la base de la nouvelle manière de modeler l'interférence. Cette thèse comble donc l'écart entre les micro-architectures modernes, qui comportent des anomalies temporelles, et les analyses d'ordonnancement, qui comptent sur la compositionalité de la réponse temporelle.

La plupart des approches modulaires pour vérifier la réponse temporelle des systèmes avec processeurs multi-cœurs calculent des bornes supérieures sur le nombre de fois un cœur peut avoir accès aux ressources partagées pendant une durée donnée. Jusqu'à présent, le calcul des bornes suppose pessimistement que le nombre d'accès aux ressources partagées par un programme puisse être distribué de manière arbitraire à l'exécution du programme. Dans cette thèse, nous proposons un calcul plus précis qui prend en compte le nombre d'accès aux ressources partagées par chacune des parties d'un programme. Le calcul proposé utilise une nouvelle variante de l'énumération implicite de chemins dans les programmes considérés. Nos expériences démontrent que le calcul nouveau donne des bornes qui sont en moyenne inférieurs d'environ cinq pour cent.

Enfin et surtout, cette thèse apporte une contribution conceptuelle à la dérivation formelle des analyses WCET. Un principe nouveau, nommé *property lifting*, permet l'utilisation des propriétés du système concret sous analyse pour la détection des comportements infaisables au niveau d'approximation sur lequel les analyses WCET fonctionnent. Nous démontrons l'application du property lifting pendant la dérivation formelle de nos analyses WCET pour systèmes avec processeurs multi-cœurs et bus partagés. Ce nouveau principe peut être également utilisé pour la dérivation formelle d'un large éventail d'approches existantes: Celles qui emploient des contraintes additionnelles pour omettre des comportements infaisables pendant le calcul d'une borne WCET. C'est une propriété intéressante dans la mesure où la plupart des approches existantes n'ont été dérivées que sur la base de l'intuition.

Ausfihrlich Sesammenfassung

Engebettete Echtzeid-Computer-Systeme fir sichahätskritische Aanwennungsbereiche – wie zum Beischpill de Ensatz in Automobilen – missen bezichlich ihrem Zeidverhallen strenge Aanforderungen erfillen. Dòòfor is de Verifizierung von em Zeidverhallen en wichtijer Dääl von da Verifizierung von de Systemen. De Verifizierung von em Zeidverhallen gefft normaalaweis in zwo Änzelschritte unnadäält. Daerscht bestimmt en Analys von da maximalen Ausfihrungszeid (*Worst-Case-Execution-Time*-Analys, WCET-Analys) pro Programm en owere Schrank von de michlijen Ausfihrungszeiden. Aanschließend nutzt en Planbarkäätsanalys de resultierend Meng von Zeidschranken um feschdseschdellen, ob in Sesammenhang mit em verwennten Ablaafplanungsalgorithmus off jeden Fall all Aanforderungen hinsichtlich em Zeidverhallen erfillt genn. De voorlijend Aawend befasst sich hauftsächlich mit da WCET-Analys.

In Mehrkeerenprozessoren genn änije Hardware-Komponenten (z.B. Iwwatrarungsweje oda Zwischenspeicha) von mehreren Prozessorkeeren sesammen genutzt um de Preis, et Gewicht un de Stromverbrauch von de Prozessoren se senken. Wejen disen Vordäälen genn Mehrkeerenprozessoren ach imma effta in engebetteten Echtzeid-Computer-Systemen fir sichahätskritische Aanwennungen engesetzt. De sesammen genutzten Komponenten machen de WCET-Analys awa vill schwierijer, weil wejen denen de Ausfihrungszeid von enem off äänem Keeren ausgefihrten Programm von de gleichzeididich off de anneren Keeren ausgefihrten Programmen abhängt. Mir bezeichnen dat als *Interferenz wejen sesammen genutzten Komponenten.* En genaue Betrachtung von disa Interferenz während da WCET-Analys wär et selwe wie de Offzehlung alla michlijen Reijenfoljen von Zougriffen off de sesammen genutzten Komponenten von de verschiedenen Prozessorkeeren un hat sich als se offwännig erwies. Dòòfor setzen de määschden existierenden Verfahren fir de Verifizierung von em Zeidverhallen von Mehrkeerenprozessoren off en modulareret Vorgehen, dat awwa zu ena niddrijeren Präzision fihrt. Zousätzlich verlòssen sich all dise modularen Verfahren (määschdens sogar nur implizit) off en *Kompositionalität von em Zeidverhallen.* Bisher is awwa net bekannt, wie solch zeidkompositionale Verfahren fir de Verifizierung von Computer-Systemen mit Zeidanomalien genutzt genn kinnen ohne dabei michlijerweis de tatsächlich Ausfihrungszeid von enem Programm se unnaschätzen. Aktuell Forschungsergebnisse lejen nah, dass sogar relativ enfache Mikroarchitekturen, die in disem Sesammenhang bisher als unproblematisch gegolt han, Zeidanomalien offweisen kinnen.

De voorlijend Aawend schläät en nauartije Modellierung von da Interferenz während da WCET-Analys fir Mehrkeerenprozessoren vor. Da vorgestellte Aansatz ermichlijt en modularet Vorgehen, weil jeweils nur et Ausfihrungsverhallen von äänem Prozessorkeeren gleichzeididich betracht gefft, un fihrt ach fir Computer-Systeme mit Zeidanomalien zu korrekten Ergebnissen. Dòriwwa hinaus zeit dise Aawend, wie en zeidkompositionalet Verfahren off Basis von da vorgestellten Modellierung umgesetzt genn kann. Off dise Weis schlää】 dise Aawend en Breck zwischen modernen Mikroarchitekturen, die Zeidanomalien offweisen kinnen, un de existierenden Planbarkäätsanalysen, die sich all off de Kompositionalität von em Zeidverhallen verlòssen.

De määschden modularen Verfahren fir de Verifizierung von em Zeidverhallen von Computer-Systemen mit Mehrkeerenprozessoren berechnen außerdem owere Schranken von de michlijen Anzahlen von Zougriffen off de sesammen genutzten Komponenten in Abhängigkät von da Läng von em jeweils betrachteten Zeidintervall. Bisher geht de Berechnung von solchen Schranken pessimistisch davon aus, dass de Gesamtzahl an Zougriffen ener Programmausfihrung beliewig iwwa de Programmausfihrung verdäält genn kann. In da voorlijenden Aawend stellen ma en feinkörnijere Berechnung vor, die droff offpasst, welcha Dääl von enem Programm welche Meng an Zougriffen off de sesammen genutzten Komponenten macht. De vorgestellte Berechnung is implementiert durch en nauartije Variant von da impliziten Pfadoffzehlung, die iwwa all teilweisen Pfade durch de betrachteten Programme argumentiert. Uusa Experimente zeijen, dass de vorgestellte Berechnung im Schnitt ungefähr fünf Prozent genauer is als de existierenden Aansätz.

Außerdem leischt de voorlijend Aawend en konzeptionellen Beitrach zu da formalen Herleitung von WCET-Analysen. Et nauartije Prinzip *Property-Lifting* ermichlijt de Verwennung von Eijenschaften von enem konkreten Computer-System fir de Identifizierung von net tatsächlich michlijen Ausfihrungsverhallen off da Abschtraktionsewene, off der WCET-Analysen argumentieren. Mir veraanschaulichen de Verwennung von em Property-Lifting im Zure von da formalen Herleitung von uusra WCET-Analys fir Mehrkeerenprozessoren mit sesammen genutzten Iwwatrarungswejen. Dòriwwa hinaus kann Property-Lifting fir de formale Herleitung von ena brääden Palett an existierenden WCET-Analysen verwennt genn, die zousätzliche Ungleichungen verwennen fir de Präzision von da berechneten Zeidschrank se erhejen. Dise existierenden WCET-Analysen sinn bisher nett off formale Weis hergeleit genn, sonnern ledichlich basierend off Beischpillen un Intuitionen.

Acknowledgments

First of all, I'd like to thank my advisor Prof. Sebastian Hack for giving me the chance to work in his group and the freedom to pursue my very own research ideas. The discussions with him were always highly constructive. I appreciate him for being a professional and—at the same time—managing to stay authentic.

Second, I want to thank my dear friend and colleague Sebastian Hahn for countless hours of enjoyable and inspiring discussions. He and Sebastian Hack permanently supported me in various ways. Without their ongoing support, this thesis would not have been possible.

Moreover, I'd like to thank all of my friends and colleagues for their support—in particular the following people and their respective partners and families: Alex, Andrea, Andreas, Angela, Anika, Arsène, Axel, Aziza, Carmen, Christian, Christoph, Claus, Daniel, Darshit, Dennis, Dominik, Eva, Fabian, Felipe, Florian, Fritz, Gabriel, Ilie, Ilina, Jan, Jochen, Johannes, Jörg, Kevin, Klaas, Knobi, Laura, Marko, Max, Mihail, Mohamed, Nadja, Nils, Pascal, Peter, Ragnar, Ralf, Richard, Roland, Sandra, Sebastian, Sigurd, Simon, Sotaya, Stefan, Steffen, Sven, Thomas, Tina, Tomasz, and Wolfgang.

In addition, I have to thank Prof. Wolfgang Paul. The formalism that he used in his lecture about computer architecture was a great source of inspiration for this thesis. I'd also like to express my gratitude to Jörg Herter, Prof. Reinhard Brocks, and Prof. Damian Weber for supporting my application at Saarland University. Furthermore, I want to thank the colleagues at the companies that I worked for. In this context, I'd like to especially thank Bernd Pfeil and Michael Schlicker.

Finally, I want to thank Prof. Christine Rochange and Prof. Reinhard Wilhelm for serving as referees on my thesis committee.

I'd like to dedicate this thesis to my grandmother Cilia.

This work has been supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS) and by the Saarbrücken Graduate School of Computer Science which receives funding from the DFG as part of the Excellence Initiative of the German Federal and State Governments.

Contents

Part I.

Overview

Chapter 1 Introduction

Welcome to the jungle. We got fun and games.
We got everything you want. Honey, we know the names.
We are the people that can find whatever you may need.
If you got the money, honey, we got your disease.

(Welcome to the Jungle, Guns N' Roses, 1987)

1.1. Timing Verification

Timing-critical computer systems must deliver the results of their computations in a timely manner. Safety-critical embedded systems—as e.g. found in the automotive, aeronautics, and industrial automation domains—are often timing-critical in the sense that a deadline miss would have catastrophic consequences. Thus, it is of utmost importance to guarantee that these systems cannot miss their deadlines. The task of providing such guarantees is referred to as *timing verification*.

The timing verification of a computer system is typically realized as a two-step approach. First, a *worst-case execution time* (WCET) analysis is performed per program executed on the considered system. It determines an upper bound on the possible execution times—a so-called WCET bound—per program under analysis. Subsequently, a *schedulability analysis* uses the WCET bounds of all programs to verify that—in combination with the processor scheduling strategy used—no program will miss its deadline.

1.2. Worst-Case Execution Time Analysis

The WCET of a program depends on the micro-architecture of the hardware platform it is executed on. The exact determination of the WCET is in general undecidable as it would solve the halting problem. Thus, WCET analysis typically resorts to the determination of an upper bound on the possible execution times that occur when executing a particular program on a particular hardware platform. Such an upper bound is referred to as *WCET bound*. Note that a WCET analysis may fail to calculate a (finite) WCET bound for a particular program executed on a particular hardware platform (either because there is no upper bound on the execution times—i.e. the exact WCET is not defined—or because the analysis is not precise enough to calculate a finite WCET bound).

There are different paradigms of WCET analysis. *Measurement-based* WCET analysis tries to derive a WCET bound based on a limited set of execution-time measurements. For realistic hardware platforms, however, it is unclear whether the actual worst case was observed during a sequence of measurements. To compensate for a potential underestimation, a safety margin is typically added to the observed worst case. It, however, remains completely unclear how to safely determine such a margin for a given hardware platform. Thus, so far, this paradigm of WCET analysis does not provide the high degree of confidence desirable for the timing verification of safety-critical systems.

Static WCET analysis, in contrast, computes invariants about all possible concrete traces of the program under analysis when executed on the considered hardware platform. As a consequence, it results in safe WCET bounds. This thesis considers the derivation and implementation of static WCET analyses.

The complexity of modern hardware designs renders an explicit consideration of all concrete traces of a program at the level of the micro-architecture infeasible in terms of analysis runtime and memory consumption. Thus, static WCET analysis typically approximates the concrete traces of the system under analysis by the abstract traces of an abstract model. This approximation has to be sound in the sense that a WCET bound calculated on the abstract model must upper-bound every possible execution time of the program under analysis on the concrete system.

There is also the relatively new paradigm of *probabilistic* WCET analysis. It aims at determining a probability distribution of WCET bounds for a program executed on a particular hardware platform. There are static and measurement-based variants of probabilistic WCET analysis. A detailed discussion of probabilistic approaches, however, exceeds the scope of this thesis. In the following, we only consider static non-probabilistic WCET analysis.

1.3. Multi-Core Processors

Multi-core processors share common resources (like e.g. caches and buses) between multiple processor cores. Thus, the manufacturers of safety-critical embedded systems plan to use multi-core processors as execution platforms instead of multiple single-core processors in order to reduce the overall cost, weight, and energy consumption. This is further encouraged by an inevitable trend in the embedded-processor market to mostly produce multi-core processors in the future.

From a timing-verification point of view, however, multi-core processors are significantly more challenging than single-core processors: the concrete traces that a program executed on a particular processor core exhibits at the micro-architectural level depend on the programs simultaneously executed on the concurrent cores [Abel et al., 2013]. This phenomenon is typically referred to as *shared-resource interference*. An exact consideration of all such interference effects would require the enumeration of all interleavings of access requests by the different cores to the shared resources—which is intractable for hardware platforms of realistic complexity. Thus, the main additional challenge in designing a WCET analysis for multi-core processors is to consider the shared-resource interference at an appropriate level of approximation.

In order to avoid the complexity of considering all interleavings of access requests, most existing approaches to WCET analysis for multi-core processors are *processor-core-modular* in the sense that they only consider one processor core at a time. First, a cumulative characterization of the shared-resource demand is determined per core. Subsequently, a WCET bound for a processor core is determined based on the cumulative demand characterizations of the concurrent cores. This effectively reduces the complexity at the cost of less precise WCET bounds.

Additionally, all existing processor-core-modular approaches rely on the principle of *timing compositionality* [Hahn et al., 2013] in the following way: they start from a WCET bound assuming the absence of shared-resource interference and subsequently add a fixed penalty per unit of concurrent interference. It, however, remains an open problem how to determine a safe interference penalty for a hardware platform that exhibits timing anomalies [Lundqvist and Stenstrom, 1999]. Our group has recently shown that shared-resource interference can trigger timing anomalies already on hardware platforms with surprisingly simple processor core pipelines, which were previously believed to be anomaly-free [Hahn et al., 2016a]. Thus, it is unclear how to safely apply the existing analyses to the majority of real-world hardware platforms.

1.4. Contributions of this Thesis

The set of all concrete traces that a particular program exhibits when executed on a particular hardware platform is in most cases too large for an explicit consideration. Thus, WCET analysis typically operates on an *abstract model* of the concrete system under analysis. An abstract trace of the abstract model can describe multiple concrete traces of the concrete system. The abstract model is sound with respect to the concrete system if its abstract traces describe all concrete traces of the concrete system. The abstract model is chosen in a way that its number of abstract traces is significantly lower than the number of concrete traces of the concrete system. Intuitively, the abstract model approximates away some of the details of the concrete system. In this way, the complexity of WCET analysis (in terms of runtime and memory consumption) is reduced to a manageable level. Relying on approximation by an abstract model, however, typically comes at the price of a reduced precision compared to directly considering the concrete system. In particular, the abstract model might feature abstract traces that do not describe any concrete traces of the concrete system. Such abstract traces are referred to as *infeasible abstract traces*. Literature proposes to use properties of the concrete system (as e.g. flow facts or cache persistence) in order to prune some of the infeasible abstract traces of the abstract model and, thus, to potentially improve the precision of the WCET bound calculated based on the remaining abstract traces of the abstract model. The use of system properties for the detection of infeasible abstract traces, however, has so far mostly been based on intuition. We close this (widely ignored) gap with our work: we present a criterion for safely *lifting properties* of the concrete system to the abstract model. As a consequence of this criterion, any abstract trace of the abstract model for which a lifted system property does not hold is guaranteed to be infeasible. Moreover, in this thesis, we formalize the concrete traces of a state-based system as well as the abstract models for the three common levels of approximation typically used in WCET analysis: micro-architectural analysis, paths through a graph, and implicit path enumeration. The resulting formal framework is used during the derivation of a WCET analysis for lifting system properties up the hierarchy of abstract models. The lifted versions of the system properties are used during the derived analysis for pruning infeasible abstract traces.

In addition, this thesis makes contributions in the area of WCET analysis for multi-core processors with shared buses. In particular, we propose an abstract model that models shared-bus interference by non-determinism and, thus, safely accounts for the effect that a unit of interference can have on the pipeline of the interfered processor core. This way of modeling is processor-core-modular and—at the same time—supports hardware platforms exhibiting timing anomalies. We derive *co-runner-insensitive* and *co-runner-sensitive* WCET analyses by lifting properties of bus arbitration policies to the proposed abstract model. A naive implementation of the non-determinism, however, leads to a significant increase of the analysis runtime compared to an analysis completely ignoring the interference. In the implementation part of this thesis, we present simple implementation tricks that avoid most of this runtime overhead.

Moreover, this thesis contributes to the field of timing compositionality. It describes a novel decomposition of the execution time into an interference-dependent part (assuming a statically configured penalty per unit of interference) and a base component (accounting for the execution time not covered by the interference-dependent part). We present the calculation of an upper bound on the base components of all concrete traces. It is referred to as *compositional base bound*. A compositional base bound can safely be used in all existing approaches to timing verification that rely on timing compositionality—even for hardware platforms that exhibit timing anomalies. Thus, the novel concept of compositional base bounds closes the gap between modern hardware platforms, which typically exhibit timing anomalies, and existing schedulability analyses, which (often only implicitly) rely on timing compositionality. The calculation of a compositional base bound relies on an analysis that safely models all possible interference effects on the considered pipeline (as e.g. the proposed way of modeling the shared-bus interference by non-determinism).

Last but not least, this thesis discusses the calculation of values on arrival curves. A value on an arrival curve upper-bounds the amount of interference that a processor core can generate in any time interval of at most a given length. Values on arrival curves are used as interference bounds in many processor-core-modular approaches to co-runner-sensitive timing verification for multi-core processors. This thesis presents a novel method for calculating values on arrival curves. The experimental evaluation demonstrates that the values calculated by the novel method are more precise than those calculated by existing methods.

Note that some of the results and contributions of this thesis have already been presented at workshops and conferences [Jacobs, 2013; Jacobs et al., 2015, 2016; Hahn et al., 2016a].

1.5. Structure of this Thesis

In Chapter 2, we sketch the state of the art in timing verification for multi-core processors by discussing the existing work in this field. Chapter 3 provides a more detailed overview of the main contributions that this thesis makes. Chapter 4 introduces the principle of property lifting. In Chapter 5, we formalize abstract models for the three levels of approximation of WCET analysis. Based on the formalized hierarchy of abstract models and the principle of property lifting, Chapter 6 demonstrates the calculation of event bounds for programs. A WCET bound is a special case of such an event bound. In Chapter 7, we present WCET analyses for multi-core processors with shared buses. These analyses model the shared-bus interference by non-determinism. Chapter 8 demonstrates the calculation of compositional base bounds. In Chapter 9, we present implementation tricks that keep the analysis-runtime overhead of modeling shared-bus interference by non-determinism at a manageable level. Chapter 10 demonstrates the calculation of values on arrival curves for characterizing the bus accesses of a processor core. Finally, Chapter 11 concludes the thesis.

Chapter 2 State of the Art in Timing Verification for Multi-Core Processors

I'm alive, but I am broken
I'm alive, but I'm ashamed

(Into the Wild, Johnossi, 2013)

This chapter sketches the state of the art in timing verification for multi-core processors by discussing the existing work in this field. The existing work related to the various other contributions that this thesis makes is discussed ad hoc in the respective chapters.

2.1. Partitioning of Shared Resources

Resources shared between multiple processor cores are often partitioned in order to reduce or completely eliminate the interference between the cores. Such a partitioning is typically static in the sense that it does not change during the execution of the system.

A static partitioning of all shared resources of a system can be used to establish *temporal isolation* [Bui et al., 2011; Perret et al., 2016] between the processor cores. Temporal isolation means that there is no shared-resource interference between the processor cores—i.e. the execution time of a program executed on one core does not depend on the programs executed at the same time on the concurrent cores. Consequently, temporal isolation enables the precise timing verification of a processor core without having to consider the programs executed on the concurrent cores. Note that temporal isolation has been a key concept during the design of a precision-timed (PRET, [Edwards and Lee, 2007]) multi-threaded processor [Liu et al., 2012].

Static partitioning schemes of shared resources have a significant impact on the (average-case as well as worst-case) performance of a system as they control which processor core has access to which share of a resource. As a consequence, there are various approaches trying to optimize the worst-case performance by choosing these schemes in a beneficial way [Rosen et al., 2007; Liu et al., 2010, 2011; John and Jacobs, 2014; Gan and Gu, 2015].

The exact way in which a shared resource is partitioned depends on whether it is a *space resource* or a *bandwidth resource.* Thus, in the following, each of these types of resources is discussed individually.

2.1.1. Space Resources

For space resources as caches and buffers, it is common to partition the space of the resource. For single-core processors, it has e.g. been proposed to partition the cache space and to assign each partition to a subset of the programs executed on the processor in order to reduce the interference the programs have on each other via the cache [Busquets-Mataix et al., 1997].

In a similar way, for multi-core processors, it has been proposed to statically partition the space of a shared cache and to assign each partition to a dedicated core [Suhendra and Mitra, 2008; Guan et al., 2009; Ungerer et al., 2010; Liu et al., 2010]. Intuitively, this enables the safe use of single-core cache analysis techniques.

There are hardware platforms which support to optionally only partition a part of the available cache space for dedicated access, while the remainder is shared between the cores in a dynamic fashion [Zang and Gordon-Ross, 2016]. There are, however, also hardware platforms which do not have hardware support for cache partitioning. For such platforms, cache partitioning can optionally also be realized in software [Plazar et al., 2009].

2.1.2. Bandwidth Resources

For bandwidth resources as buses and interconnects, it is common to partition the time and to assign each resulting time slot to a dedicated processor core. This is referred to as *time-division multiple access* (TDMA). In recent years, there have been various approaches to timing verification that support TDMA bus arbitration [Lv et al., 2010; Schranzhofer et al., 2010a, 2011; Kelter et al., 2011; Chattopadhyay et al., 2012; Kelter et al., 2014; Altmeyer et al., 2015; Rihani et al., 2015].

Note that there are hardware platforms which support to optionally only configure some of the available time slots for dedicated access, while the remainder is shared between the cores in a dynamic fashion [Schranzhofer et al., 2011].

Modern processors—even in the embedded domain—are typically designed for a good average-case performance. Thus, they typically only support event-driven bus arbitration policies. As a consequence, approaches to timing verification relying on TDMA bus arbitration are only of limited use for modern multi-core processors.

2.2. Unpartitioned Shared Resources

For unpartitioned shared resources, it is in some cases possible to bound the amount of interference experienced by one processor core independently of the programs executed on the concurrent cores. This is also referred to as *Murphy* approach [Abel et al., 2013; Hahn et al., 2016a] as it implicitly assumes the concurrent cores to execute programs that maximize the interference on the core under analysis. For shared caches, it is always sound to pessimistically assume that each access might miss or hit the shared cache [Yan and Zhang, 2008]. For some shared-bus arbitration protocols (e.g. *Round-Robin*), the interference can be bounded independently of the programs on the concurrent cores [Jacobs et al., 2015]. For other shared-bus arbitration protocols (e.g. *priority-based*, which may exhibit starvation), this is not possible.

The Murphy approach is considered [Paolieri et al., 2009] as the preferred approach to timing verification for hard real-time threads executed on the multi-core architecture developed in the MERASA project [Ungerer et al., 2010].

The Murphy approach has been shown to lead to significant overestimation of the interference [Yan and Zhang, 2008; Nowotsch, 2014; Jacobs et al., 2015]. Thus, most existing approaches take into account the resource access behavior of the concurrent cores. There are roughly two classes of approaches, which differ in the degree of detail at which the interference between the programs on the different cores is modeled.

2.2.1. Enumeration of Interleavings of Access Requests

The enumeration of all interleavings of access requests to the shared resources enables the precise bounding of the impact of shared-resource interference on the execution time of a program. This way of modeling shared-resource interference is also referred to as *fully integrated* [Hahn et al., 2016a].

Kelter and Marwedel present a micro-architectural analysis that explores all interleavings of access requests [Kelter and Marwedel, 2014]. For a multi-core processor with a shared bus and an event-driven bus arbitration, their analysis is on average 130 times slower than an existing analysis considering TDMA bus arbitration [Chattopadhyay et al., 2012]. This dramatic increase in analysis runtime is observed although they assume that the programs executed on the different processor cores start perfectly synchronized. In his dissertation [Kelter, 2015], Kelter also evaluates an analysis that models shared-cache interference in the same way. For a dual-core scenario, he reports an average analysis runtime of 1,489.9 seconds per benchmark, compared to 2 seconds for an existing, less precise approach [Li et al., 2009]. He does not present experiments considering the combined modeling of a shared bus and a shared cache. In our opinion, this way of modeling shared-resource interference cannot scale to real-world scenarios.

The use of timed model checking [Bengtsson et al., 1995] for the enumeration of all interleavings of access requests to the shared resources has also been proposed [Lv et al., 2010; Gustavsson et al., 2010; Giannopoulou et al., 2012; Lampka et al., 2014]. However, it suffers from similar scalability problems as the approach by Kelter and Marwedel. Ironically, the authors of one of these papers claim that their approach "scales well and can handle real-life programs" [Lv et al., 2010] in combination with a shared bus and a *first-come-first-serve* bus arbitration. This claim is not supported at all by the corresponding experiments in the paper, which only consider a dual-core platform. Two of these papers [Giannopoulou et al., 2012; Lampka et al., 2014] can be seen as the more integrated counterparts of an earlier processor-core-modular algebraic method [Schranzhofer et al., 2011]. Surprisingly, however, the approaches in both papers are not compared to the earlier algebraic method or any other existing approach—neither in terms of runtime nor in terms of precision.

Zhang and Yan present an ILP-based approach that models shared-cache interference by arguing about all processor cores in an integrated fashion [Zhang and Yan, 2009]. They only conduct experiments for a dual-core processor with a direct-mapped shared cache. Their approach is based on an earlier ILP-based approach to cache modeling [Li et al., 1996], which is known for its high computational complexity [Lv et al., 2016]. Thus we are confident, that their approach does not scale to realistic multi-core platforms.

Although this line of work does not seem to scale to real-world scenarios, it can serve—in combination with toy scenarios—as a benchmark for evaluating the precision of more scalable approaches.

2.2.2. Processor-Core-Modular Timing Verification

In order to overcome the seemingly inherent scalability problems of modeling shared-resource interference in a fully integrated way, most existing approaches resort to more modular timing verification techniques [Schliecker et al., 2008; Yan and Zhang, 2008; Negrean et al., 2009; Li et al., 2009; Schliecker et al., 2009; Hardy et al., 2009; Andersson et al., 2010; Schliecker and Ernst, 2010; Schranzhofer et al., 2010b; Chattopadhyay et al., 2010; Pellizzoni et al., 2010; Schranzhofer et al., 2011; Dasari et al., 2011; Dasari and Nélis, 2012; Nowotsch and Paulitsch, 2013; Nowotsch et al., 2014; Nowotsch, 2014; Nagar and Srikant, 2014; Altmeyer et al., 2015; Nagar and Srikant, 2016; Huang et al., 2016; Wegener, 2017]. These approaches are *processor-core-modular* in the sense that they only consider one processor core at a time. A cumulative characterization of the shared-resource access behavior is determined per processor core. The actual timing verification of a particular processor core relies on the access characterizations of the concurrent processor cores. The improved scalability of the processor-core-modular approaches comes at the cost of a reduced precision [Kelter and Marwedel, 2014]. Thus, processor-core-modular approaches might not be able to verify some of the timing properties that can be verified with fully integrated approaches.

Note that all aforementioned processor-core-modular approaches additionally rely on the principle of *timing compositionality* [Hahn et al., 2013]. They typically start from a WCET bound assuming the absence of shared-resource interference and subsequently add a fixed penalty pen_R per unit of concurrent interference on shared resource R. The determination of safe interference penalties for the different shared resources of a hardware platform that exhibits timing anomalies [Lundqvist and Stenstrom, 1999], however, remains an open problem. Our group has recently shown that shared-resource interference can trigger timing anomalies already on hardware platforms with surprisingly simple processor core pipelines [Hahn et al., 2016a], which were previously believed to be anomaly-free. Thus, it is unclear how to safely apply the existing analyses to the majority of real-world hardware platforms.

2.2.3. Relevance of this Thesis

This thesis proposes to model the shared-bus interference by non-determinism and, thus, to safely account for the effect that a unit of interference can have on the pipeline of the interfered processor core. This way of modeling enables a processor-core-modular consideration and—at the same time—supports hardware platforms exhibiting timing anomalies. Based on this way of modeling, we present a *co-runner-insensitive* WCET analysis (corresponding to the Murphy approach) and a *co-runner-sensitive* WCET analysis (considering the bus access behavior of the programs on the concurrent processor cores).

Moreover, this thesis describes a novel decomposition of the execution time into an interference-dependent part (assuming a statically configured penalty per unit of interference) and a base component (accounting for the execution time not covered by the interference-dependent part). We present the calculation of an upper bound on the base components of all concrete traces. It is referred to as *compositional base bound*. A compositional base bound can safely be used in all existing approaches to timing verification that rely on timing compositionality—even for hardware platforms that exhibit timing anomalies. Thus, this novel concept makes the aforementioned processor-core-modular approaches applicable to a significantly increased range of hardware platforms. The calculation of a compositional base bound relies on an analysis that safely accounts for all possible interference effects on the considered pipeline (as e.g. the proposed way of modeling the shared-bus interference by non-determinism).

Note that—to the best of our knowledge—we are first to recognize the common level of modularity that we refer to as *processor-core-modularity*. The term processor-core-modularity is inspired by *thread-modularity*, which refers to verification techniques for multi-threaded software that only consider one thread at a time and characterize the write behavior with respect to the shared variables in a cumulative way per thread [Flanagan and Qadeer, 2003; Henzinger et al., 2003; Malkis et al., 2007; Gotsman et al., 2007; Miné, 2011, 2014; Monat and Miné, 2017].

Chapter 3 Contributions

There is a war between us
I wanna give you something
There is a force that drives us

(The War between Us, Tiger Lou, 2004)

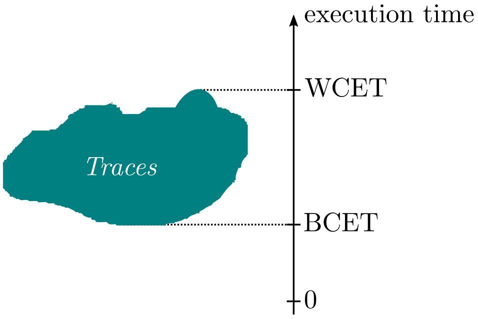
This chapter provides a brief overview of the contributions that this thesis makes with respect to the state of the art. For more details, we refer to the following chapters.

Note that some of the results and contributions of this thesis have already been presented at workshops and conferences [Jacobs, 2013; Jacobs et al., 2015, 2016; Hahn et al., 2016a].

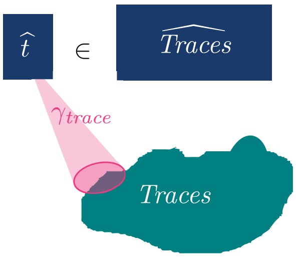
3.1. Property Lifting

In this thesis, we present the principle of *property lifting*. It uses properties of a concrete system under analysis to prune infeasible abstract traces from an abstract model of the concrete system. The formal details of property lifting are presented in Chapter 4. In the following, we use a sequence of figures to intuitively explain how property lifting works.

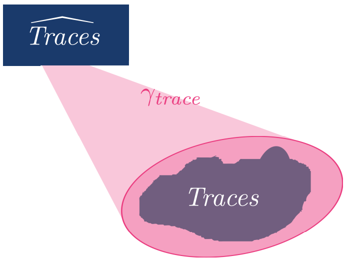
We start by taking into account the set *Traces* of *concrete traces* of the concrete system under analysis. In the context of timing verification, a concrete trace is typically a sequence of hardware states that corresponds to an execution of the program under analysis on the concrete system. Each concrete trace has an *execution time*. The *worst-case execution time* (WCET) is the maximal execution time over all concrete traces. Analogously, the *best-case execution time* (BCET) is the minimal execution time over all concrete traces.



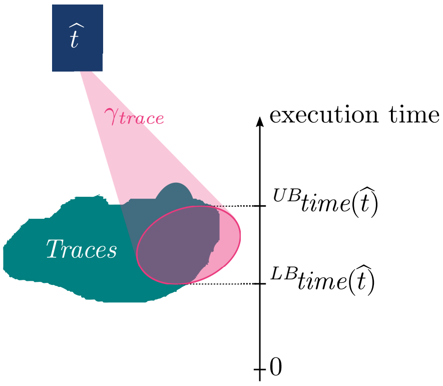
For modern hardware platforms, the set of all concrete traces is typically too large to explicitly consider each concrete trace separately. Thus, we need an *abstract model* approximating away some of the details of the concrete system. The abstract model consists of a set $\widehat{Traces}$ of abstract traces and a description function γ_{trace}. Each abstract trace $\widehat{t}$ describes a set of concrete traces. However, since this way of approximation is not necessarily precise, an abstract trace $\widehat{t}$ might also describe *spurious traces* (i.e. traces that are not contained in the set *Traces* of the concrete system).



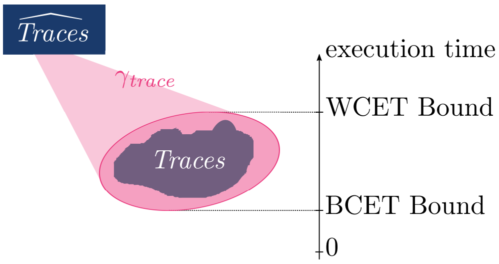
In order to provide a sound abstract model of the concrete system, each concrete trace has to be described by an abstract trace of $\widehat{Traces}$.



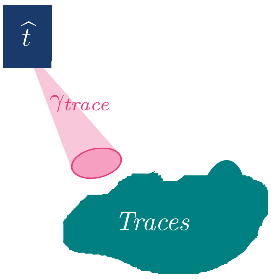
Each abstract trace has an upper bound and a lower bound on the execution times of the concrete and spurious traces that it describes.



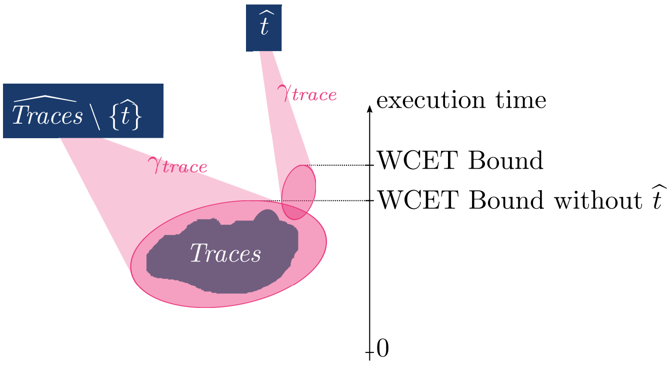
A WCET bound is obtained by choosing the maximum upper time bound over all abstract traces. Analogously, a BCET bound is obtained by choosing the minimum lower time bound over all abstract traces.



Remember that we only require that each concrete trace has to be described by at least one abstract trace of the abstract model. Thus, the abstract model might as well feature abstract traces that do not describe any concrete trace. We call them *infeasible* abstract traces.

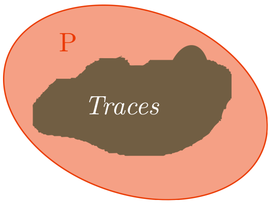


Unfortunately, the WCET bound might be dominated by infeasible abstract traces. Thus, we have a vital interest in *pruning infeasible abstract traces* in order to *improve the precision* of the WCET bound.

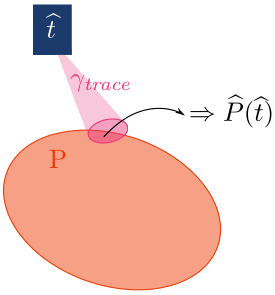


Recall that we use an abstract model in order to not have to argue explicitly about all concrete traces. Thus, it is no option to check the feasibility of an abstract trace by explicitly considering what it describes.

Instead, we propose to use *system properties* for the detection of infeasible abstract traces. A system property P of a considered concrete system is a Boolean predicate that holds for each concrete trace of the concrete system. However, it might also hold for some of the spurious traces described by an abstract model of the concrete system.

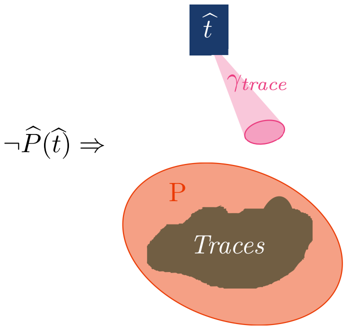


A system property P is not of direct use for the detection of infeasible abstract traces as it is so far only defined on the sets of concrete and spurious traces. Thus, we define a lifted version $\widehat{P}$ of the system property as a Boolean predicate on abstract traces. The lifted property $\widehat{P}$ has to fulfill a *soundness criterion* with respect to the system property P: if an abstract trace $\widehat{t}$ describes something for which the system property P holds, the lifted property $\widehat{P}$ also has to hold for $\widehat{t}$.



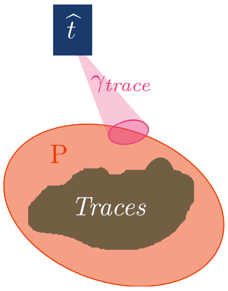
Note that—for an *efficient implementation*—it is mandatory that the lifted property $\widehat{P}$ can be evaluated without having to explicitly consider the traces described by the respective abstract trace.

Now, assume that the lifted property $\widehat{P}$ does not hold for a particular abstract trace $\widehat{t}$. As a consequence of the soundness criterion, $\widehat{t}$ can only describe traces for which property P does not hold. It follows from P being a system property that $\widehat{t}$ cannot describe any concrete traces of the concrete system. Thus, $\widehat{t}$ is infeasible and can safely be pruned.



This gives an intuition that property lifting is indeed sound. We present a more formal soundness argument in Chapter 4.

However, note that property lifting is *not necessarily complete*. This means that it does not necessarily detect all infeasible abstract traces. As an example, consider the following system property P, which cannot be used to detect the infeasibility of abstract trace $\widehat{t}$. Any safely lifted version $\widehat{P}$ of P will fail to detect $\widehat{t}$ as infeasible.



In this thesis, we lift properties of shared-bus arbitration protocols (cf. Chapter 7) to a hierarchy of abstract models corresponding to the levels of approximation used in WCET analysis (cf. Chapter 5). Note, however, that property lifting is a very general principle, which does not make any assumptions about the structure of the concrete traces or the abstract traces. Thus, its application is not limited to timing verification. In principle, it can as well be applied to a wide range of verification techniques relying on approximation (as e.g. *predicate abstraction* [Graf and Saïdi, 1997] or *shape analysis via three-valued logic* [Sagiv et al., 2002]). A detailed discussion of such applications of property lifting, however, is beyond the scope of this thesis. In Section 4.2.5, we argue that an iterative variant of property lifting (cf. Section 4.2) can as well be used for the derivation of thread-modular verification techniques [Flanagan and Qadeer, 2003] for multi-threaded software.

3.2. Formalization of the Levels of Approximation

The typical WCET analysis workflow (i.e. a micro-architectural analysis and a subsequent implicit path enumeration) can be seen as operating on three levels of approximation.

1. At the level of approximation of *micro-architectural analysis*, there are sequences of abstract states [Thesing, 2004].

2. At the level of approximation of *paths through a graph*, there are the paths through a directed, weighted graph [Matthies, 2006; Stein, 2010].

3. At the level of approximation of *implicit path enumeration*, there are so-called implicit paths. An implicit path approximates away the order of the edges along a path and only counts how often an edge occurs [Li and Malik, 1995; Puschner and Schedl, 1997; Theiling, 2002; Stein, 2010].

In Chapter 5, we formalize an abstract model for each of these three levels of approximation. Figure 3.1 shows a schematic overview of the resulting hierarchy of abstract models. Each abstract model is formally shown to describe all traces of the underlying level.

Due to each abstract model in the hierarchy safely describing all traces of the level below, we can use the principle of property lifting in order to lift system properties up the hierarchy. In this way, we can use the lifted system properties in order to safely detect and prune infeasible abstract traces of the abstract models at the different levels of approximation.

The idea of using additional knowledge about a concrete system in order to prune infeasible abstract traces from an abstract model of the concrete system is not new. There are various approaches that add additional linear constraints to an implicit path enumeration based on ILP in order to improve the precision of the resulting WCET bounds [Li et al., 1995, 1996; Englbom

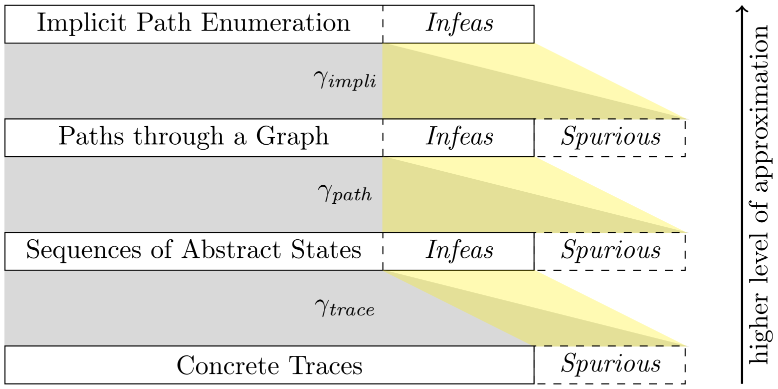


Figure 3.1.: Schematic overview of the hierarchy of abstract models:
▨ Each *feasible* abstract trace describes some feasible traces of the level below.
🟨 An *infeasible* abstract trace does not describe any feasible trace of the level below.

and Ermedahl, 2000; Stein, 2010; Cullmann, 2013]. The derivation of these approaches, however, was mostly based on intuition. Consequently, there are so far no formal soundness proofs for most of these approaches.

Thus, our work closes a gap that has been widely ignored: Our framework enables a formally sound derivation of the constraints that are added to an implicit path enumeration in order to improve the analysis precision. We are confident that the aforementioned existing approaches can be formalized as instances of our framework and proved sound in this way.

For a more detailed discussion on the relevance of this formal hierarchy and the existing work related to it, we refer to Section 5.5.

3.3. Calculation of General Event Bounds

A *system event* is an entity that happens during some of the clock cycles of the execution of a hardware platform. Thus, a system event is precisely defined by the set of cycle transitions of the hardware during which it happens. System events are e.g. cache misses, cycles blocked at a shared bus, and writes to a particular memory location. Note that the *cycle tick event* happens during any clock cycle of the hardware platform.

A *WCET bound* of a particular program must never be exceeded by the number of cycle tick events during any execution run of the program on the system under analysis. We can generalize this requirement to a whole class of upper event bounds by replacing the cycle tick event by arbitrary other system events.

In Chapter 6, we present the calculation of such general event bounds based on the hierarchy of abstract models and a set of lifted system properties. Moreover, we formally prove the soundness of the resulting bounds.

Existing approaches to the calculation of WCET bounds typically only consider paths from the start to the end of the program under analysis. For the calculation of general event bounds, this results in safe bounds for programs that are guaranteed to terminate (cf. Section 6.4.7). This approach, however, is not necessarily sound for the calculation of general event bounds for programs that can diverge.

Consider Figure 3.2 for two counter-examples. It depicts the control flow graphs of two example programs that shall be able to diverge. For simplicity, we assume that every execution of a particular basic block of one of the programs exhibits the constant number of occurrences of event E_1 annotated to the basic block. Moreover, each basic block shall be reached by at least one execution run of the respective program.

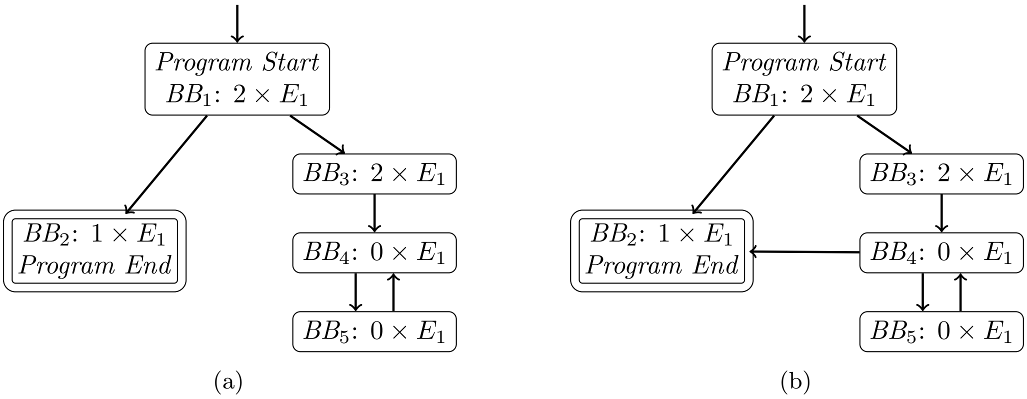


Figure 3.2.: Control flow graphs of two example programs that can diverge. For simplicity, the execution of each basic block is assumed to exhibit a constant number of occurrences of event E_1.

For the example program in Figure 3.2a, the consideration of all paths from the program start to the program end results in a bound value of three with respect to the number of occurrences of event E_1. This bound value is not sound as there are four occurrences of event E_1 on a path from the program start to the loop.

Now, consider the example program in Figure 3.2b. The only difference to the example program in Figure 3.2a is that—according to the control flow graph—the program end is also reachable from within the loop. For this variant of the example program, the set of all paths from the program start to the program end results in a bound value of five with respect to the number of occurrences of event E_1. This bound value is sound as there cannot be more than five occurrences of event E_1 during an execution of the example program.

Finally, assume we additionally know that the example program in Figure 3.2b cannot reach the program end once it entered the loop. This might e.g. be based on the results of a sophisticated flow analysis which finds out that the execution of basic block BB_4 is always succeeded by the execution of basic block BB_5. Consequently, we use this knowledge as system property in order to prune all paths that reach the program end from within the loop. The remaining set of paths from the program start to the program end results in a bound value of three with respect to the number of occurrences of event E_1. Again, this bound value is not sound as there are four occurrences of event E_1 on a path from the program start to the loop. This means that—in combination with property lifting—the reachability of the program end from every basic block of the control flow graph is not sufficient for a safe calculation of a general event bound based on the paths from the program start to the program end.

We propose to overcome these soundness issues by exploiting the graph theory concept of *feedback node sets* [Karp, 1972]. A feedback node set of a graph is a subset of its nodes chosen in a way that every cycle in the graph contains a node of the subset. Once we determined a feedback node set of the graph, we calculate an event bound based on all paths from the programs start to either the program end or one of the feedback nodes. In Section 6.4.3, we formally prove that this approach is sound for arbitrary programs—independent of their termination behavior.

Intuitively, this means that we simply add a valid set of feedback nodes of the graph to its set of end nodes. For the example programs in Figure 3.2, it is e.g. sufficient to add basic block BB_4 to the set of end nodes of the graph. The consideration of all paths from the program start to either the program end or BB_4 results in a bound value of four, which is sound with respect to the number of occurrences of event E_1.

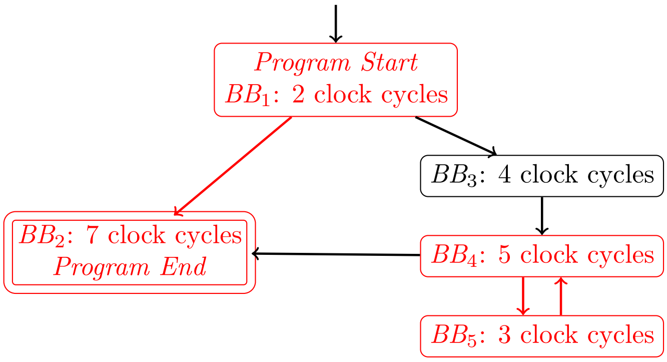


Figure 3.3.: Control flow graph of another example programs that can diverge. For simplicity, the execution of each basic block is assumed to take a constant amount of clock cycles.

Our feedback node set approach never calculates a defined event bound value that is unsound with respect to the concrete program. The classical approach of only considering paths from the program start to the program end, in contrast, must be combined with a termination analysis in order to provide the same guarantee.

Even in combination with a termination analysis, the classical approach cannot guarantee any defined bound value if the termination analysis cannot prove the termination of the program. The feedback node set approach, in contrast, can calculate defined event bound values for programs that can diverge. This is also demonstrated by the example program in Figure 3.2a. The feedback node set approach succeeds to calculate a defined bound value of four. The combination of the classical approach and a termination analysis, in contrast, has to pessimistically assume an unbounded number of event occurrences.

Note that the classical approach suffers from similar soundness problems during the calculation of lower event bounds—although there are no such soundness problems for the special case of BCET bounds (cf. equation (6.32)). In Chapter 6, we prove the soundness of the feedback node set approach for the calculation of upper and lower event bounds.

Further note that the classical approach suffers from similar soundness problems during the calculation of WCET bounds at the level of approximation of paths through a graph. However, for the special case of WCET bounds, the soundness is guaranteed to be reestablished at the level of approximation of implicit path enumeration in case the implicit path enumeration is based solely on linear constraints (i.e. no additional features of modern solvers like SOS constraints or indicator constraints are used). We would like to demonstrate this with the example in Figure 3.3. It depicts the control flow graphs of an example program that shall be able to diverge. For simplicity, we assume that every execution of a particular basic block of the program takes a constant amount of clock cycles. Moreover, each basic block shall be reached by at least one execution run of the concrete program. Assume that we additionally know that the example program cannot reach the program end once it executes BB_3. This flow fact is expressed as a linear constraint during implicit path enumeration as follows.

$$\mathit{timesTaken}_{BB_3} + \mathit{timesTaken}_{BB_2} \leq 1$$

If we perform a calculation of a WCET bound at the level of approximation of paths through a graph by only considering paths from the program start to the program end and only considering paths in which basic blocks BB_3 and BB_2 do not both occur, we obtain a bound value of nine clock cycles, which is not sound for a program that can diverge. At the level of approximation of implicit path enumeration, however, the objective is unbounded in case the implicit path enumeration is based solely on linear constraints. The reason for this is that, in implicit path enumeration based solely on linear constraints, any unbounded loop in the graph can hold itself within an implicit path without necessarily having to be connected to any of the remaining nodes and edges of the implicit path. In Figure 3.3, we color the nodes and edges that belong to the implicit paths leading to an unbounded objective. In Section 6.5.2, we argue that such implicit paths always exist for a diverging program in case the implicit path enumeration is based solely on linear constraints.

3.4. Modeling Shared-Resource Interference by Non-Determinism

The micro-architectural analysis of a given program for a given hardware platform *typically* assumes the absence of shared-resource interference. The resulting time bounds are used by subsequent analyses [Schliecker et al., 2008; Yan and Zhang, 2008; Negrean et al., 2009; Li et al., 2009; Schliecker et al., 2009; Hardy et al., 2009; Andersson et al., 2010; Schliecker and Ernst, 2010; Schranzhofer et al., 2010b; Chattopadhyay et al., 2010; Pellizzoni et al., 2010; Schranzhofer et al., 2011; Dasari et al., 2011; Dasari and Nélis, 2012; Nowotsch and Paulitsch, 2013; Nowotsch et al., 2014; Nowotsch, 2014; Nagar and Srikant, 2014; Altmeyer et al., 2015; Nagar and Srikant, 2016; Huang et al., 2016; Wegener, 2017] which add a fixed temporal penalty per unit of concurrent interference (e.g. per cycle of being blocked at a shared bus). The determination of such safe penalties for hardware platforms exhibiting timing anomalies, however, remains an open problem [Hahn et al., 2016a].

There are approaches [Gustavsson et al., 2010; Kelter and Marwedel, 2014] that take into account the shared-resource interference during micro-architectural analysis. In general (i.e. without necessarily relying on a partitioning of the shared resources, cf. Section 2.1), however, these approaches rely on an enumeration of all interleavings of access requests by the different processor cores. Experimental results indicate that this enumeration suffers from a very high computational complexity [Kelter, 2015]. Thus, such a fully integrated analysis of the shared-resource interference is not expected to scale to real-world scenarios (cf. Section 2.2.1).

In this thesis, we propose to model shared-resource interference by *non-determinism* during micro-architectural analysis. In contrast to a fully integrated consideration of the shared-resource interference, the proposed approach only explicitly argues about the program execution on a single processor core. The single considered processor-core pipeline is argued about in a similar way as in single-core WCET analysis [Thesing, 2004]. The access behavior of the considered processor core with respect to the shared resources, however, is pessimistically overapproximated by non-deterministic case splits. Figure 3.4 demonstrates this principle for a system with a shared bus and a fixed access latency of two clock cycles, which every granted bus access takes. The non-determinism pessimistically assumes that every requested bus access might be blocked for an arbitrary number of clock cycles before it is granted. Section 7.2 describes this non-deterministic overapproximation of the shared-bus interference in a more detailed way.

In Section 7.8, we briefly sketch how this non-deterministic overapproximation of the shared-bus interference can be extended to additionally support further sources of interference (as e.g. shared caches and/or DRAM refreshes). Such further sources of interference, however, are not in the focus of this thesis. Thus, the analysis derivations and case studies presented in this thesis assume that a shared bus is the only source of interference with respect to the considered processor core.

Note that the non-deterministic overapproximation of the shared-bus interference as presented so far (cf. Figure 3.4) is still relatively useless for WCET analysis. It pessimistically assumes an unbounded amount of blocked cycles per bus access. Thus, in order to obtain finite WCET bounds, we exploit system properties which upper-bound the number of blocked cycles. Lifted versions (cf. Section 3.1) of these system properties are used for the safe detection of infeasible abstract traces. For the derivation of co-runner-insensitive WCET analyses, we only use system

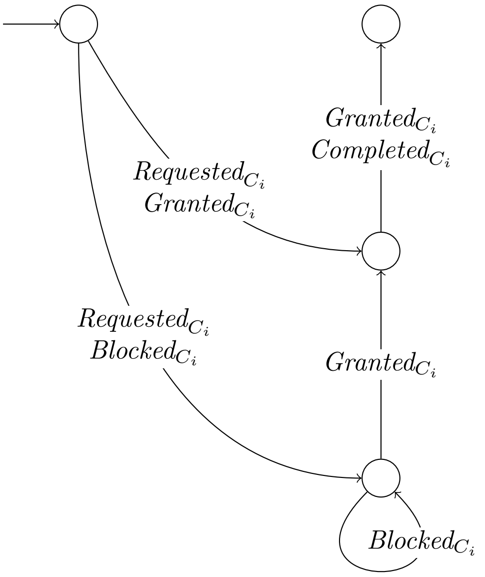


Figure 3.4.: Diagram representation of the non-determinism pessimistically modeling the access behavior with respect to a shared bus and a fixed access latency of two clock cycles.

properties which bound the number of blocked cycles independently of the programs executed on the concurrent cores (cf. Section 7.4). The derived co-runner-insensitive analyses only take into account the program and the processor core for which a WCET bound is calculated. For the derivation of co-runner-sensitive WCET analyses, we additionally use system properties which bound the number of blocked cycles in dependence of the access behavior of the concurrent cores (cf. Section 7.5). The derived co-runner-sensitive analyses are processor-core-modular and, thus, avoid the enumeration of all interleavings of access requests by the different processor cores.

To the best of our knowledge, the presented modeling scheme for shared-resource interference is the first one supporting systems with timing anomalies and at the same time not resorting to a fully integrated analysis. Thus, this modeling scheme is an important step toward the timing verification of hard real-time systems with real-world multi-core processors.

The presented modeling scheme avoids the computational complexity of a fully integrated analysis. Yet, unsurprisingly, it is computationally more complex than simply assuming the absence of interference. In Chapter 9, we demonstrate how simple implementation tricks can help keeping the overhead in terms of analysis runtime manageable.

3.5. Compositional Base Bounds

Timing-compositional [Hahn et al., 2013] analyses typically start from a basic time bound and add a fixed penalty per unit of interference. The following equation represents a simple timing-compositional analysis that calculates an upper bound on the execution times of a program in dependence on the number of cycles the execution is blocked at the shared bus.

$$Time = Base + NumberBlockedCycles \cdot Penalty$$

Classically, the bound *Base* is calculated by single-core WCET analysis techniques (i.e. assuming the absence of interference). For hardware platforms exhibiting timing anomalies [Lundqvist and Stenstrom, 1999], however, the determination of a safe penalty per unit of interference is an open problem [Hahn et al., 2016a]. Intuitively, the challenge is that—in the presence of timing anomalies—a cycle of being blocked at the shared bus might prolong the overall execution time by more than a single clock cycle. As a consequence, so far, the analyses relying on timing compositionality are not applicable to hardware platforms exhibiting timing anomalies.

In order to overcome this limitation, we propose to calculate *Base* on top of a low-level analysis that safely models the shared-bus access behavior of the considered processor core (cf. Section 3.4). Our calculation procedures assume a (manually determined) penalty per unit of interference. In this way, we are able to calculate a bound *Base* that fulfills the following intuitive *soundness statement*:

> Any concrete trace that is blocked at the shared bus for at most x cycles has an execution time of at most $(\mathit{Base} + x \cdot \mathit{Penalty})$ clock cycles.

A bound *Base* that fulfills this soundness statement with respect to a given penalty is referred to as *compositional base bound.* In Chapter 8, we formalize the concept of compositional base bounds and present procedures for their calculation. In this way, we close the gap between high-level timing analyses (as e.g. schedulability analyses) that (implicitly or explicitly) rely on timing compositionality and modern hardware platforms that typically exhibit timing anomalies.

3.6. Calculation of Values on Arrival Curves

Processor-core-modular approaches (cf. Section 2.2.2) to the timing verification of systems with multi-core processors rely on a cumulative approximation of the shared-resource access behavior per processor core. Such a cumulative approximation usually consists in calculating values on an arrival curve. A value $\alpha(l)$ on an arrival curve α upper-bounds the number of occurrences of an interference-generating event E on a given processor core in any time interval of at most l clock cycles. Such an interference-generating event might e.g. be a granted access to the shared bus by the given processor core.

The calculation of values on arrival curves is typically performed at the *granularity of program runs.* This means that the calculation pessimistically assumes the maximal amount of occurrences of event E any execution run of a given program *prog* generates to be distributable across the execution run in an arbitrary fashion. This principle is demonstrated by Figure 3.5 for a scenario in which the processor core under consideration repeatedly executes a single program *prog.* Each execution run of program *prog* is assumed to take the best-case execution time $\mathit{BCET}_{\mathit{prog}}$ and to produce the maximal amount $\mathit{Max}_{E,\mathit{prog}}$ of event occurrences. The number of occurrences of event E in a time interval of length l is maximized if the first execution run spanned by the interval generates all interference events at the end of the program and all subsequent execution runs generate all interference events at the beginning of the program.

The calculation of arrival curve values at the granularity of program runs assumes that the amount of event occurrences generated by an execution run can be distributed across the execution run in an arbitrary fashion. For most real-world programs, however, the occurrences of interference-generating events are more or less distributed across the whole execution run. Thus, a calculation at the granularity of program runs is inherently pessimistic for most real-world programs. As an example, consider a time interval of a length l that does not exceed $2 \cdot \mathit{Max}_{E,\mathit{prog}}$. According to the pessimistic assumptions of the calculation at program granularity, such an interval can be completely filled up with occurrences of event E (i.e. $\alpha(l) = l$, cf. Figure 3.6).

Another precision problem of the calculation of arrival curve values at the granularity of program runs stems from the pessimistic assumption that every execution run of a considered program only takes the BCET of the program and—at the same time—is able to generate the maximal amount of event occurrences that an execution run can generate. For most real-world programs, however, execution runs that take longer are typically also able to generate a higher amount of event occurrences. As an example, consider a program *prog* such that $\mathit{BCET}_{\mathit{prog}} \leq \mathit{Max}_{E,\mathit{prog}}$. According to the pessimistic assumptions of the calculation at program granularity, for this

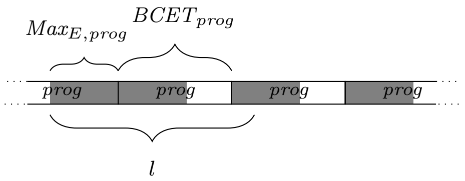


Figure 3.5.: The calculation of arrival curve values at the granularity of program runs pessimistically assumes that the maximal amount of interference an execution run of a program generates can be distributed across the execution run in an arbitrary way. Under this assumption, the amount of interference that can be generated in a time interval of length l is maximized if the first execution run spanned by the interval generates all interference events at the end of the program and all subsequent execution runs generate all interference events at the beginning of the program.

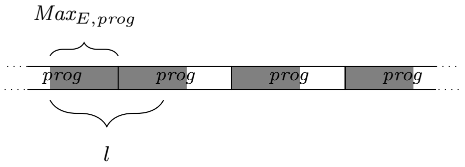


Figure 3.6.: The calculation of arrival curve values at the granularity of program runs is particularly pessimistic for time intervals of a length l that does not exceed $2 \cdot Max_{E,prog}$. According to the assumed (and typically infeasible) worst-case distribution of events, such intervals can be completely filled up with event occurrences.

example, a time interval of any length l can be completely filled up with occurrences of event E (i.e. $\forall l \in \mathbb{N} : \alpha(l) = l$). Note that, for a program with a wide range of possible executions times (i.e. $WCET_{prog}$ is significantly greater than $BCET_{prog}$), it is not unrealistic to assume $BCET_{prog} \leq Max_{E,prog}$.

Finally, we would like to point out that the calculation of arrival curve values at the granularity of program runs inherently relies on the existence of an upper bound $Max_{E,prog}$ on the number of event occurrences per execution run of each program *prog* executed on the considered processor core. For programs that can diverge, however, such a bound does typically not exist. Thus, as soon as the considered processor core executes a program that can diverge, a calculation at the granularity of program runs is not applicable. Instead, it is pessimistically assumed that any time interval can be completely filled up with event occurrences (i.e. $\alpha(l) = l$). This precision problem might e.g. arise for scenarios in which only some of the processor cores of a system execute timing-critical programs.

In order to overcome these precision problems of the calculation of arrival curve values at the granularity of program runs, in Section 10.2, we present the calculation of arrival curve values at finer granularities. The calculation is based on graphs that are similar to those typically used for the calculation of a WCET bound. These graphs are e.g. at the granularity of basic blocks or at the granularity of clock cycles. In contrast to the calculation of a WCET bound, however, the calculation of an arrival curve value also has to consider execution sequences on a processor core that span across multiple program execution runs. This is achieved by additional graph edges that connect the end nodes of a program with the start nodes of all programs executed on the same processor core. In addition, the calculation of an arrival curve value also has to consider execution sequences that start at an arbitrary point of an execution run and end at an arbitrary point of an execution run. To this end, our calculation argues about all subpaths of the underlying graph representation. In order to efficiently argue about these subpaths, we exploit the principle of implicit path enumeration—resulting in an *implicit subpath enumeration*. An experimental evaluation shows that the presented calculation results in a significantly improved precision compared to a calculation at the granularity of program runs.

The calculation of arrival curve values via implicit subpath enumeration, however, suffers from a high computational complexity. Thus, it is unlikely to scale to scenarios in which multiple real-world programs are executed per processor core. In Section 10.4, we sketch a more modular calculation of arrival curve values. It performs the computationally complex implicit subpath enumerations on a per-program level. The actual curve value calculation uses the resulting per-program compositional base bounds in a compact ILP formulation. We expect that, in this way, the modular calculation provides a precision close to the curve value calculation via implicit subpath enumeration while significantly improving on its scalability.

Part II.

Design of WCET Analyses

Chapter 4 Property Lifting

I will lift you up my friend
You have to learn to breathe again

(Monsters in the Ballroom, In Flames, 2014)

4.1. The Principle of Property Lifting

The (potentially only partial) verification of a complex computer system typically requires some degree of approximation by an abstract model of the concrete system in order to make verification tractable in terms of runtime and memory consumption. Approximation, however, typically results in a loss of precision. As a consequence, an imprecise abstract model of a concrete system may prohibit the proof of a verification goal that holds for the concrete system. Property lifting provides a mechanism to improve the precision of an abstract model by pruning some of its members that are guaranteed to describe no concrete traces of the concrete system.

This section presents the principle of property lifting at a formal level. For a less formal introduction to property lifting, we refer to Section 3.1.

4.1.1. Approximation by Abstract Traces

The term *system* refers to any formal or physical system that exhibits observable behavior. Formal systems include but are not limited to state machines and term rewrite systems. In the context of WCET analysis, we use the term system to refer to the combination of a *hardware platform and the software* executed on it. The considered concrete system may exhibit different concrete traces depending on its initial state, external input parameters and the environment. Let *Traces* be the set of all concrete traces of the system. Its superset *Universe* additionally contains the spurious traces that might be described by imprecisely approximating the concrete system. Spurious traces can, for example, be sequences of concrete system states that cannot be observed during any execution of the concrete system.

$$\mathit{Universe} \supseteq \mathit{Traces} \tag{4.1}$$

Complex systems usually exhibit too many concrete traces to allow for an exhaustive consideration of all of them. The set *Traces* is simply too large. Therefore, it is mandatory to introduce some kind of approximation. The goal is to not have to argue separately about each concrete trace.

We propose the approximation by a set $\widehat{Traces}$ of abstract traces. The function γ_{trace} maps abstract traces to subsets of concrete and spurious traces. Note that $\mathcal{P}(Universe)$ denotes the power set of the union of concrete and spurious traces.

$$\gamma_{trace} : \widehat{Traces} \to \mathcal{P}(Universe) \tag{4.2}$$

We consider the tuple $(\widehat{Traces}, \gamma_{trace})$ as an *abstract model* of the concrete system if and only if it overapproximates the set *Traces* of concrete traces.

$$\bigcup_{\hat{t} \in \widehat{Traces}} \gamma_{trace}(\hat{t}) \supseteq Traces \tag{4.3}$$

The principles presented in this chapter are applicable to any abstract model (providing the aforementioned overapproximation). Thus, for now, we do not make any assumptions about the structure of abstract traces or the exact definition of γ_{trace}. In Chapter 5, we formally define one abstract model for each of the three levels of approximation typically used in WCET analysis.

4.1.2. Infeasible Abstract Traces

Approximation typically introduces imprecision. This manifests in abstract traces describing subsets of the concrete and spurious traces. As a consequence, there may be abstract traces that do not describe any concrete trace (i.e. they only describe spurious traces or nothing at all). We call them *infeasible* abstract traces.

$$\widehat{Infeas} = \{\hat{t} \in \widehat{Traces} \mid \gamma_{trace}(\hat{t}) \cap Traces = \emptyset\} \tag{4.4}$$

Correspondingly, we refer to $\widehat{Traces} \setminus \widehat{Infeas}$ as the set of *feasible* abstract traces. In fact, it follows from (4.4) that the set of feasible abstract traces is an overapproximation of *Traces*.

$$\bigcup_{\hat{t} \in \widehat{Traces} \setminus \widehat{Infeas}} \gamma_{trace}(\hat{t}) \supseteq Traces \tag{4.5}$$

Intuitively, any subset of $\widehat{Traces}$ containing all its feasible abstract traces also provides an overapproximation of *Traces*.

$$\widehat{Traces} \supseteq \widehat{Traces}' \supseteq \widehat{Traces} \setminus \widehat{Infeas} \Rightarrow \bigcup_{\hat{t} \in \widehat{Traces}'} \gamma_{trace}(\hat{t}) \supseteq Traces \tag{4.6}$$

As a consequence, we can safely prune an arbitrarily chosen set of infeasible abstract traces in an abstract model. Pruning infeasible abstract traces can improve the precision of the abstract model, e.g. by pruning abstract traces that assume an overly high amount of execution time during WCET analysis.

We propose the approximation by abstract traces in order to not have to explicitly consider the set *Traces*. The definition of infeasible abstract traces, however, is also based on *Traces*. Therefore, we cannot directly use this definition to detect infeasible abstract traces. The following subsection describes how we can use properties of the system under consideration to safely detect infeasible abstract traces.

4.1.3. System Properties

We assume properties to be Boolean predicates on concrete and spurious traces. System properties of a concrete system are properties that hold for each concrete trace of the concrete system. The existence of a bound on the shared-resource interference may for example be a system property. Let *Prop* be a set of properties of the system under consideration.

$$Prop = \{P_1, \ldots, P_p\} \tag{4.7}$$

$$\forall t \in Traces : \forall P_k \in Prop : P_k(t) \tag{4.8}$$

We would like to use these system properties to detect (some of the) infeasible abstract traces. However, so far, they only argue about concrete and spurious traces. Therefore, we need to *lift* them to abstract traces. This means, we need to find $\widehat{P_k}$ such that the following *soundness criterion* holds.

$$\forall \widehat{t} \in \widehat{Traces} : [\, \exists t \in \gamma_{trace}(\widehat{t}) : P_k(t) \,] \Rightarrow \widehat{P_k}(\widehat{t}) \tag{4.C1}$$

The intuition behind soundness criterion (4.C1) gets more clear as soon as we take a look at what it means that $\widehat{P_k}$ does not hold for an abstract trace $\widehat{t} \in \widehat{Traces}$.

$$\begin{aligned} & \neg\widehat{P_k}(\widehat{t}) \\ \underset{(4.C1)}{\Rightarrow} \quad & \forall t \in \gamma_{trace}(\widehat{t}) : \neg P_k(t) \\ \underset{(4.8)}{\Rightarrow} \quad & \gamma_{trace}(\widehat{t}) \cap Traces = \emptyset \\ \underset{(4.4)}{\Leftrightarrow} \quad & \widehat{t} \in \widehat{Infeas} \end{aligned} \tag{4.9}$$

So if a lifted system property does not hold for an abstract trace, this means that the abstract trace is infeasible. From now on, the lifted version of any system property P_k shall be identified by $\widehat{P_k}$.

4.1.4. Pruning Infeasible Abstract Traces

We define a compound property $\widehat{P}$ for abstract traces to be the conjunction over the lifted versions of the considered system properties.

$$\begin{aligned} & \forall \widehat{t} \in \widehat{Traces} : \\ & \quad \widehat{P}(\widehat{t}) \Leftrightarrow \forall P_k \in Prop : \widehat{P_k}(\widehat{t}) \end{aligned} \tag{4.10}$$

An abstract trace $\widehat{t}$ for which $\widehat{P}$ does not hold is infeasible.

$$\begin{aligned} & \neg\widehat{P}(\widehat{t}) \\ \underset{(4.10)}{\Leftrightarrow} \quad & \exists P_k \in Prop : \neg\widehat{P_k}(\widehat{t}) \\ \underset{(4.9)}{\Rightarrow} \quad & \widehat{t} \in \widehat{Infeas} \end{aligned} \tag{4.11}$$

Thus, we use $\widehat{P}$ to define an alternative set $\widehat{LessTraces}$ of abstract traces based on $\widehat{Traces}$.

$$\widehat{LessTraces} = \{\widehat{t} \mid \widehat{t} \in \widehat{Traces} \wedge \widehat{P}(\widehat{t})\} \tag{4.12}$$

$\widehat{LessTraces}$ is the subset of abstract traces in $\widehat{Traces}$ that is not detected as infeasible by any of the $\widehat{P_k}$. Thus, it is an overapproximation of the feasible abstract traces.

$$\widehat{LessTraces} \supseteq \widehat{Traces} \setminus \widehat{Infeas} \tag{4.13}$$

According to equations (4.6), (4.12), and (4.13), $(\widehat{LessTraces}, \gamma_{trace})$ overapproximates $Traces$ and, thus, also is an abstract model of the concrete system.

$$\bigcup_{\widehat{t} \in \widehat{LessTraces}} \gamma_{trace}(\widehat{t}) \supseteq Traces \tag{4.14}$$

$(\widehat{LessTraces}, \gamma_{trace})$ can improve the precision, as the set $\widehat{LessTraces}$ potentially prunes some of the infeasible abstract traces still included in $\widehat{Traces}$. In that context, $(\widehat{Traces}, \gamma_{trace})$ is referred to as *baseline abstract model* as it is the starting point for further improvements of precision.

This concludes the description of the concept of property lifting. Intuitively, the main idea is to start from a pessimistic abstract model as baseline. Lifted versions of system properties are used to detect some infeasible abstract traces of the baseline abstract model. Pruning them may result in a more precise abstract model.

4.1.5. Related Work

In the context of timing verification for multi-core processors, there are multiple approaches [Schranzhofer et al., 2011; Giannopoulou et al., 2012; Chattopadhyay et al., 2012; Kelter and Marwedel, 2014] that do not only support a single hardware platform but whole classes of micro-architectures. Each of these approaches is parametric in one or more aspects of the supported hardware platform (e.g. processor core pipeline [Chattopadhyay et al., 2012] or bus arbitration [Schranzhofer et al., 2011; Giannopoulou et al., 2012; Kelter and Marwedel, 2014]). As discussed in Chapter 2, none of these approaches is applicable to current multi-core processors.

The property lifting framework, in contrast, is the first generic framework for the formal derivation of timing analyses for multi-core processors. It is not restricted to a particular analysis paradigm. This thesis presents its instantiation based on a state-of-the-art WCET analysis [Thesing, 2004; Stein, 2010]. In principle, it could as well be used for the derivation of e.g. timed automata [Alur and Dill, 1990; Giannopoulou et al., 2012]. Although property lifting was introduced in the context of timing verification [Jacobs, 2013], its application is not limited to this field of research. In principle, it can as well be applied to a wide range of verification techniques relying on approximation (as e.g. *predicate abstraction* [Graf and Saïdi, 1997] or *shape analysis via three-valued logic* [Sagiv et al., 2002]). A detailed discussion of such applications of property lifting, however, is beyond the scope of this thesis.

The theory of abstract interpretation [Cousot and Cousot, 1977] enables the creation of sound abstract models of the semantics of computer programs by the construction of fixed points of equation systems on ordered domains. The principle of property lifting, in contrast, relies on the existence of a (potentially imprecise) baseline abstract model as starting point. The abstract traces of the baseline abstract model do not have to belong to an ordered domain as no fixed point is constructed. Instead, property lifting provides a mechanism for safely using system properties of the concrete system to detect infeasible abstract traces of the baseline abstract model. In this thesis, the fixed point of an abstract-interpretation-based micro-architectural analysis [Thesing, 2004] is used as baseline abstract model for the property lifting framework (cf. Section 5.2).

The key principle of property lifting is the usage of knowledge about a concrete system in order to improve the precision of an abstract model of the concrete system by pruning abstract traces that do not describe any concrete trace. This principle is also employed in a recent ILP-based approach to bound the blocking experienced by a task due to spin locks in a shared resource setting [Wieder and Brandenburg, 2013]. The simple baseline model makes rather pessimistic assumptions about the blocking. More detailed knowledge about the possible sources of blocking and feasible combinations of them can be formulated as additional constraints in order to improve the blocking bound.

The idea of starting from an imprecise model and subsequently improving its precision is common to abstraction refinement [Clarke et al., 2000] as applied to model checking. Abstraction refinement aims at successively specifying the model at an increasingly fine granularity in order to verify properties that cannot be verified at the initial granularity. The property lifting framework, in contrast, operates on an abstract model at a given fixed granularity. It improves the precision by pruning abstract traces that do not describe any concrete trace.

4.2. Cooperation between Multiple Abstract Models

In order to verify the correct operation of a complex system, it is common to consider multiple abstract models that focus on different aspects of the concrete system's operation (e.g. on different hardware components of the system). Such a *compositional consideration* avoids the complexity of modeling the interactions between the different aspects of the concrete system in detail. However, it typically results in a loss of precision.

In the context of WCET analysis for multi-core processors, each abstract model typically only considers the operation of one processor core in detail. This avoids the complexity of an exhaustive enumeration of all possible interleavings of accesses to the shared resources by the different processor cores. As a further consequence, this typically results in a loss of precision for platforms that do not provide temporal isolation between their processor cores.

Most existing approaches to WCET analysis for multi-core processors further decompose the execution time into non-interfered execution (e.g. assuming every access request to a shared bus is granted immediately by the arbiter) and direct interference effects (e.g. the number of cycles blocked at a shared bus). As pointed out in Chapter 2, such a decomposition is not sound for hardware platforms that exhibit timing anomalies as it ignores potential indirect interference effects.

Approaches to timing verification that rely on a decomposition of the execution time are referred to as *timing-compositional* [Hahn et al., 2013]. Note that the formalism derived in this section is not primarily designed to be applicable in the context of timing compositionality. Instead, it is closely related to the principles behind thread-modular analyses and verification techniques. Thus, in the remainder of this section, the term compositionality does not necessarily coincide with the term timing compositionality.

In the following, we formally define a compositional consideration by multiple abstract models that focus on different parts of the concrete system. In the model of one part of the system, it is often desirable to incorporate (mostly cumulative) information about other parts of the system, e.g. high-level information about the resource access behavior of concurrent processor cores. The property lifting framework supports this exchange of information between multiple abstract models by leveraging system properties that interrelate the operation of different parts of the system. To this end, first, we define a compound abstract model based on a set of abstract models. System properties interrelating the operation of multiple parts of the system are lifted to the compound abstract model in order to prune infeasible compound abstract traces. Finally, we present iterative algorithms that safely overapproximate the results of this compound consideration without actually having to explicitly consider the compound abstract model. This provides a trade-off between the precision of the compound abstract model and the efficiency of using its component abstract models in isolation.

4.2.1. A Compound Abstract Model

Let *Models* be a set of abstract models of the concrete system under consideration.

$$\mathit{Models} = \{M_1, \ldots, M_m\} \tag{4.15}$$

$$\forall M_a \in \mathit{Models} : M_a = (\widehat{\mathit{Traces}}^{M_a}, \gamma_{\mathit{trace}}^{M_a}) \tag{4.16}$$

The different abstract models focus on different aspects of the concrete system. In the context of WCET analysis for multi-core processors, each abstract model might, e.g., focus on a different processor core.

Based on the abstract-trace sets of previous abstract models, we can define a set $\widehat{\mathit{Traces}}$ of compound abstract traces.

$$\widehat{\mathit{Traces}} = \widehat{\mathit{Traces}}^{M_1} \times \ldots \times \widehat{\mathit{Traces}}^{M_m} \tag{4.17}$$

We use projection functions $\pi_{\mathit{trace}}^{M_a}$ to access the components of compound abstract traces.

$$\begin{gathered}\forall(\widehat{t^{M_1}}, \ldots, \widehat{t^{M_m}}) \in \widehat{\mathit{Traces}}^{M_1} \times \ldots \times \widehat{\mathit{Traces}}^{M_m} : \\ \forall M_a \in \mathit{Models} : \pi_{\mathit{trace}}^{M_a}((\widehat{t^{M_1}}, \ldots, \widehat{t^{M_m}})) = \widehat{t^{M_a}}\end{gathered} \tag{4.18}$$

The mapping of a compound abstract trace to concrete and spurious traces can be defined as the intersection of what its component abstract traces describe. Intuitively, a compound abstract trace only describes the traces that all of its components describe.

$$\gamma_{\mathit{trace}}(\widehat{t}) = \bigcap_{M_a \in \mathit{Models}} \gamma_{\mathit{trace}}^{M_a}(\pi_{\mathit{trace}}^{M_a}(\widehat{t})) \tag{4.19}$$

It follows from equations (4.17) and (4.19) that the tuple $(\widehat{\mathit{Traces}}, \gamma_{\mathit{trace}})$ provides an overapproximation of *Traces* and, thus, is a (compound) abstract model of the concrete system under consideration. Hence, we can apply property lifting to it as demonstrated in Section 4.1. Lifted properties can detect infeasible compound abstract traces and, thus, result in a reduced set $\widehat{\mathit{LessTraces}}$.

In order to visualize this concept, Figure 4.1 presents an example of a compound abstract model. It is based on the two abstract models A and W arguing about three respectively four abstract traces. Thus, the compound abstract model argues about twelve compound abstract traces. Subsequently, we lift the two system properties P_1 and P_2 to the compound abstract model. For our simple example, it does not matter which statements these system properties make about the traces for which they hold. The lifted version of each property shall hold for five of the compound abstract traces. The set $\widehat{\mathit{LessTraces}}$ only contains the three compound abstract traces for which the lifted versions of both system properties hold.

However, the cross product in the definition of $\widehat{\mathit{Traces}}$ already gives a hint that $\widehat{\mathit{Traces}}$ might become quite large. Thus, we expect the compound consideration of multiple abstract models to be computationally too complex for most real-world scenarios.

4.2.2. Projections of the Compound Results

We introduced the compound abstract model in order to profit in one abstract model from knowledge about other abstract models and, thus, to detect more infeasible abstract traces per abstract model. As a consequence, in most cases, it would be sufficient to know for each $M_a \in \mathit{Models}$ which members of $\widehat{\mathit{Traces}}^{M_a}$ are contained in a compound abstract trace of $\widehat{\mathit{LessTraces}}$. Those subsets can be obtained by projecting the members of $\widehat{\mathit{LessTraces}}$ to their

$$\mathit{Models} = \{A, W\}$$

$$\widehat{\mathit{Traces}}^{A} = \{\widehat{t_a}, \widehat{t_b}, \widehat{t_c}\} \qquad \widehat{\mathit{Traces}}^{W} = \{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}, \widehat{t_z}\}$$

$$\widehat{\mathit{Traces}} = \widehat{\mathit{Traces}}^{A} \times \widehat{\mathit{Traces}}^{W} \qquad \text{cf. (4.17)}$$

$$\mathit{Prop} = \{P_1, P_2\}$$

$$\{\widehat{t} \in \widehat{\mathit{Traces}} \mid \widehat{P_1}(\widehat{t})\} = \{(\widehat{t_a}, \widehat{t_w}), (\widehat{t_a}, \widehat{t_x}), (\widehat{t_a}, \widehat{t_y}), (\widehat{t_b}, \widehat{t_x}), (\widehat{t_c}, \widehat{t_y})\}$$

$$\{\widehat{t} \in \widehat{\mathit{Traces}} \mid \widehat{P_2}(\widehat{t})\} = \{(\widehat{t_a}, \widehat{t_w}), (\widehat{t_a}, \widehat{t_x}), (\widehat{t_b}, \widehat{t_x}), (\widehat{t_b}, \widehat{t_y}), (\widehat{t_c}, \widehat{t_z})\}$$

$$\forall \widehat{t} \in \widehat{\mathit{Traces}} : \widehat{P}(\widehat{t}) \Leftrightarrow \widehat{P_1}(\widehat{t}) \wedge \widehat{P_2}(\widehat{t}) \qquad \text{cf. (4.10)}$$

$$\widehat{\mathit{LessTraces}} = \{(\widehat{t_a}, \widehat{t_w}), (\widehat{t_a}, \widehat{t_x}), (\widehat{t_b}, \widehat{t_x})\} \qquad \text{cf. (4.12)}$$

Figure 4.1.: Example of a compound abstract model based on the two abstract models A and W.

respective components. We define the projections in a general way on arbitrary subsets $\widehat{\mathit{SomeTraces}}$ of $\widehat{\mathit{Traces}}$.

$$\forall M_a \in \mathit{Models} : \pi^{M_a}(\widehat{\mathit{SomeTraces}}) = \{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{SomeTraces}}\} \tag{4.20}$$

Each projection $\pi^{M_a}(\widehat{\mathit{SomeTraces}})$ is a subset of the set of abstract traces of the corresponding abstract model.

$$\forall \widehat{\mathit{SomeTraces}} \subseteq \widehat{\mathit{Traces}} : \forall M_a \in \mathit{Models} : \pi^{M_a}(\widehat{\mathit{SomeTraces}}) \subseteq \widehat{\mathit{Traces}}^{M_a} \tag{4.21}$$

Note that $\widehat{\mathit{SomeTraces}}$ is a subset of the cross product over its projections.

$$\forall \widehat{\mathit{SomeTraces}} \subseteq \widehat{\mathit{Traces}} : \widehat{\mathit{SomeTraces}} \subseteq \pi^{M_1}(\widehat{\mathit{SomeTraces}}) \times \ldots \times \pi^{M_m}(\widehat{\mathit{SomeTraces}}) \tag{4.22}$$

Furthermore, note that the projection functions π^{M_a} are monotone.

$$\begin{gathered} \forall \widehat{\mathit{SomeTraces}}, \widehat{\mathit{OtherTraces}} \subseteq \widehat{\mathit{Traces}} : \forall M_a \in \mathit{Models} : \\ [\, \widehat{\mathit{SomeTraces}} \subseteq \widehat{\mathit{OtherTraces}} \,] \Rightarrow [\, \pi^{M_a}(\widehat{\mathit{SomeTraces}}) \subseteq \pi^{M_a}(\widehat{\mathit{OtherTraces}}) \,] \end{gathered} \tag{4.23}$$

Each projection $\pi^{M_a}(\widehat{\mathit{LessTraces}})$ is a superset of the feasible abstract traces of the corresponding $\widehat{\mathit{Traces}}^{M_a}$. Consult page 247 for a formal proof of this statement.

$$\forall M_a \in \mathit{Models} : \pi^{M_a}(\widehat{\mathit{LessTraces}}) \supseteq \widehat{\mathit{Traces}}^{M_a} \setminus \widehat{\mathit{Infeas}}^{M_a} \tag{4.24}$$

Thus, according to equations (4.6), (4.21) and (4.24), each tuple $(\pi^{M_a}(\widehat{\mathit{LessTraces}}), \gamma^{M_a}_{\mathit{trace}})$ provides an overapproximation of *Traces* and, thus, is an abstract model of the concrete system.

$$\forall M_a \in \mathit{Models} : \bigcup_{\widehat{t^{M_a}} \in \pi^{M_a}(\widehat{\mathit{LessTraces}})} \gamma^{M_a}_{\mathit{trace}}(\widehat{t^{M_a}}) \supseteq \mathit{Traces} \tag{4.25}$$

The projections of the set $\widehat{\mathit{LessTraces}}$ to the abstract models A and W for the example of Figure 4.1 are presented in Figure 4.2. The tuples $(\pi^{A}(\widehat{\mathit{LessTraces}}), \gamma^{A}_{\mathit{trace}})$ and $(\pi^{W}(\widehat{\mathit{LessTraces}}), \gamma^{W}_{\mathit{trace}})$ are abstract models of the concrete system considered in this example.

$$\pi^{A}(\widehat{\mathit{LessTraces}}) = \{\widehat{t_a}, \widehat{t_b}\} \qquad \pi^{W}(\widehat{\mathit{LessTraces}}) = \{\widehat{t_w}, \widehat{t_x}\} \qquad \text{cf. (4.20)}$$

Figure 4.2.: Projections of the set $\widehat{\mathit{LessTraces}}$ to the abstract models A and W for the example presented in Figure 4.1.

However, in most cases we will not be able to precisely obtain the projections $\pi^{M_a}(\widehat{\mathit{LessTraces}})$ without explicitly computing the set $\widehat{\mathit{LessTraces}}$ before. We expect the computation of the set $\widehat{\mathit{LessTraces}}$ to be computationally too expensive. Therefore, we are interested in overapproximations of its projections that can be computed more efficiently.

4.2.3. Overapproximating the Projections by Starting from a Maximally Pessimistic Initialization

We present an iterative approach that overapproximates each projection $\pi^{M_a}(\widehat{\mathit{LessTraces}})$ by a corresponding approximation variable $\widehat{\mathit{Approx}}^{M_a}$. In the following, we use $\overrightarrow{\mathit{Approx}}$ as a shorthand for the vector of all approximation variables. In the same way, we use $\overrightarrow{\mathit{Approx}_{\overline{M_a}}}$ as a shorthand for the vector of all approximation variables except $\widehat{\mathit{Approx}}^{M_a}$.

$$\overrightarrow{\mathit{Approx}} \equiv (\widehat{\mathit{Approx}}^{M_1}, \ldots, \widehat{\mathit{Approx}}^{M_m}) \tag{4.26}$$

$$\forall M_a \in \mathit{Models} : \overrightarrow{\mathit{Approx}_{\overline{M_a}}} \equiv (\widehat{\mathit{Approx}}^{M_1}, \ldots, \widehat{\mathit{Approx}}^{M_{a-1}}, \widehat{\mathit{Approx}}^{M_{a+1}}, \ldots, \widehat{\mathit{Approx}}^{M_m}) \tag{4.27}$$

The set $\widehat{\mathit{Traces}}^{M_a}$ is a safe overapproximation of the projection $\pi^{M_a}(\widehat{\mathit{LessTraces}})$. Thus, we use $\widehat{\mathit{Traces}}^{M_a}$ as a safe but *maximally pessimistic initialization* of approximation variable $\widehat{\mathit{Approx}}^{M_a}$.

$$\forall M_a \in \mathit{Models} : \widehat{\mathit{Approx}}^{M_a} \leftarrow \widehat{\mathit{Traces}}^{M_a} \tag{4.28}$$

Subsequently, the value of each approximation variable $\widehat{\mathit{Approx}}^{M_a}$ is updated by an *update function* F^{M_a}. The updated value depends on the contents of all other approximation variables.

$$\forall M_a \in \mathit{Models} : \widehat{\mathit{Approx}}^{M_a} \leftarrow F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}}) \tag{4.29}$$

The goal of an update is to prune abstract traces of an approximation variable while preserving the overapproximation of the corresponding projection. Intuitively, an abstract trace is a member of a projection $\pi^{M_a}(\widehat{\mathit{LessTraces}})$ if and only if it is a part of a compound abstract trace for which $\widehat{P}$ holds. Thus, an update must not prune an abstract trace of an approximation variable if the abstract trace fulfills $\widehat{P}$ in combination with members of the other approximation variables. It is essential that the update functions guarantee this.

In order to visualize this concept, we resume the example presented in Figures 4.1 and 4.2. Figure 4.3 demonstrates the iterative overapproximation of the projections $\pi^{A}(\widehat{\mathit{LessTraces}})$ and $\pi^{W}(\widehat{\mathit{LessTraces}})$ from a pessimistic initialization. Note that the update functions provide the aforementioned guarantee. Their formal derivation, however, will be explained later. For this example, we assume that the approximation variables are updated simultaneously in every iteration. The table in Figure 4.3 shows the contents of the approximation variables after the initialization as well as after every iteration. The initialization is maximally pessimistic (cf. equation (4.28)). The approximation variables reach a fixed point after the second iteration.

Update functions:

$$F^A(\widehat{Approx}^W) = \{\widehat{t^A} \in \widehat{Traces}^A \mid [\exists \widehat{t^W} \in \widehat{Approx}^W : \widehat{P_1}((\widehat{t^A}, \widehat{t^W}))] \wedge \exists \widehat{t^W} \in \widehat{Approx}^W : \widehat{P_2}((\widehat{t^A}, \widehat{t^W}))\}$$

$$F^W(\widehat{Approx}^A) = \{\widehat{t^W} \in \widehat{Traces}^W \mid [\exists \widehat{t^A} \in \widehat{Approx}^A : \widehat{P_1}((\widehat{t^A}, \widehat{t^W}))] \wedge \exists \widehat{t^A} \in \widehat{Approx}^A : \widehat{P_2}((\widehat{t^A}, \widehat{t^W}))\}$$

Simultaneous update per iteration:

$$(\widehat{Approx}^A, \widehat{Approx}^W) \leftarrow (F^A(\widehat{Approx}^W), F^W(\widehat{Approx}^A))$$

Course of iteration:

iteration	$\widehat{Approx}^A$	$\widehat{Approx}^W$
init	$\{\widehat{t_a}, \widehat{t_b}, \widehat{t_c}\}$	$\{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}, \widehat{t_z}\}$
1	$\{\widehat{t_a}, \widehat{t_b}, \widehat{t_c}\}$	$\{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}\}$
2	$\{\widehat{t_a}, \widehat{t_b}\}$	$\{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}\}$
3	$\{\widehat{t_a}, \widehat{t_b}\}$	$\{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}\}$

Figure 4.3.: Iterative overapproximation of the projections $\pi^A(\widehat{LessTraces})$ and $\pi^W(\widehat{LessTraces})$ presented in Figure 4.2. The iteration starts from a pessimistic initialization.

Their contents at the fixed point provide an overapproximation of the projections of $\widehat{LessTraces}$ (cf. Figure 4.2). Note, however, that the content of $\widehat{Approx}^W$ at the fixed point is not precise with respect to $\pi^W(\widehat{LessTraces})$.

The decision whether a particular abstract trace should be kept in the updated content of a particular approximation variable depends on the abstract trace as well as on the contents of the other approximation variables. Thus, we write the update functions in a generic form and factor out this decision to a Boolean predicate $\widetilde{P^{M_a}}$.

$$\forall M_a \in Models : F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) = \{\widehat{t^{M_a}} \in \widehat{Traces}^{M_a} \mid \widetilde{P^{M_a}}(\widehat{t^{M_a}}, \overrightarrow{Approx_{\overline{M_a}}})\} \tag{4.30}$$

We mentioned before that an update function F^{M_a} must not prune an abstract trace if there is a combination with abstract traces of the other approximation variables that fulfills $\widehat{P}$. Now, we formally define this requirement on $\widetilde{P^{M_a}}$ as *soundness criterion* (4.C2).

$$\begin{aligned} &\forall M_a \in Models : \forall \widehat{t^{M_a}} \in \widehat{Traces}^{M_a} : \forall \overrightarrow{Approx_{\overline{M_a}}} : \\ &\quad [\exists (\widehat{t^{M_1}}, \ldots, \widehat{t^{M_{a-1}}}, \widehat{t^{M_{a+1}}}, \ldots, \widehat{t^{M_m}}) \in \chi(\overrightarrow{Approx_{\overline{M_a}}}) : \\ &\quad\quad \widehat{P}(\widehat{t^{M_1}}, \ldots, \widehat{t^{M_{a-1}}}, \widehat{t^{M_a}}, \widehat{t^{M_{a+1}}}, \ldots, \widehat{t^{M_m}})] \\ &\quad \Rightarrow \widetilde{P^{M_a}}(\widehat{t^{M_a}}, \overrightarrow{Approx_{\overline{M_a}}}) \end{aligned} \tag{4.C2}$$

Note that $\chi(\overrightarrow{Approx_{\overline{M_a}}})$ is a shorthand for the cross product over all approximation variables in $\overrightarrow{Approx_{\overline{M_a}}}$.

$$\begin{aligned} &\forall M_a \in Models : \\ &\chi(\overrightarrow{Approx_{\overline{M_a}}}) \equiv \widehat{Approx}^{M_1} \times \ldots \times \widehat{Approx}^{M_{a-1}} \times \widehat{Approx}^{M_{a+1}} \times \ldots \times \widehat{Approx}^{M_m} \end{aligned} \tag{4.31}$$

Generic form of F^A:

$$F^A(\widehat{Approx}^W) = \{\widehat{t^A} \in \widehat{Traces}^A \mid \widetilde{P^A}(\widehat{t^A}, \widehat{Approx}^W)\}$$

Formal derivation of $\widetilde{P^A}$ via a chain of implications:

$$\begin{aligned}
&[\exists \widehat{t^W} \in \widehat{Approx}^W : \widehat{P}(\widehat{t^A}, \widehat{t^W})] \\
\Leftrightarrow &[\exists \widehat{t^W} \in \widehat{Approx}^W : \widehat{P_1}(\widehat{t^A}, \widehat{t^W}) \wedge \widehat{P_2}(\widehat{t^A}, \widehat{t^W})] \\
\Rightarrow &\left[[\exists \widehat{t^W} \in \widehat{Approx}^W : \widehat{P_1}(\widehat{t^A}, \widehat{t^W})] \wedge \exists \widehat{t^W} \in \widehat{Approx}^W : \widehat{P_2}(\widehat{t^A}, \widehat{t^W})\right] \\
\Leftrightarrow: &\widetilde{P^A}(\widehat{t^A}, \widehat{Approx}^W)
\end{aligned}$$

Figure 4.4.: Formal evidence that the update function F^A presented in Figure 4.3 respects soundness criterion (4.C2). The reasoning for update function F^W is similar.

As a consequence of criterion (4.C2), each approximation variable is guaranteed to be an overapproximation of the corresponding projection after arbitrary sequences of updates of the approximation variables. This statement is captured by hypothesis (4.H1). For a detailed proof of this hypothesis, we refer to page 247.

$$\forall M_a \in Models : \widehat{Approx}^{M_a} \supseteq \pi^{M_a}(\widehat{LessTraces}) \tag{4.H1}$$

It follows from hypothesis (4.H1) and equations (4.24) and (4.6) that—at any point within the course of iteration—each tuple $(\widehat{Approx}^{M_a}, \gamma^{M_a}_{trace})$ is an overapproximation of *Traces* and, thus, an abstract model of the concrete system.

$$\forall M_a \in Models : \bigcup_{\widehat{t^{M_a}} \in \widehat{Approx}^{M_a}} \gamma^{M_a}_{trace}(\widehat{t^{M_a}}) \supseteq Traces \tag{4.32}$$

Note that the soundness of this iterative approach does not necessarily rely on a simultaneous update of all approximation variables. Any other update strategy is also applicable.

At this point, we would like to resume the example of Figure 4.3. Figure 4.4 provides formal evidence that the update function F^A of this example indeed respects soundness criterion (4.C2). The soundness argument for update function F^W is similar and, thus, omitted.

The soundness of the transfer functions is guaranteed by criterion (4.C2). In addition, however, we also require the update functions F^{M_a} to be monotone. This requirement is formalized as *monotonicity criterion* (4.C3) on the Boolean predicates $\widetilde{P^{M_a}}$.

$$\begin{aligned}
&\forall M_a \in Models : \forall \widehat{t^{M_a}} \in \widehat{Traces}^{M_a} : \forall \overrightarrow{Approx_{\overline{M_a}}}, \overrightarrow{Approx'_{\overline{M_a}}} : \\
&\overrightarrow{Approx'_{\overline{M_a}}} \subseteq_{pairwise} \overrightarrow{Approx_{\overline{M_a}}} \\
&\Rightarrow [\widetilde{P^{M_a}}(\widehat{t^{M_a}}, \overrightarrow{Approx'_{\overline{M_a}}}) \Rightarrow \widetilde{P^{M_a}}(\widehat{t^{M_a}}, \overrightarrow{Approx_{\overline{M_a}}})]
\end{aligned} \tag{4.C3}$$

Note that the example predicate $\widetilde{P^A}$ derived in Figure 4.4 trivially fulfills criterion (4.C3). This holds in the same way for the corresponding predicate $\widetilde{P^W}$ implicitly used in the update function F^W presented in Figure 4.3.

In combination with the maximally pessimistic initialization (cf. equation (4.28)), criterion (4.C3) guarantees that the updates of the approximation variables are *monotonically decreasing*. This is expressed by hypothesis (4.H2). For a formal proof of this hypothesis, we refer to page 248.

$$\forall M_a \in \mathit{Models} : F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}}) \subseteq \widehat{\mathit{Approx}}^{M_a} \tag{4.H2}$$

As a consequence, the presented iterative approach is guaranteed to reach a fixed point after a finite sequence of updates of the approximation variables if each approximation variable is initialized to a finite set of abstract traces (i.e. $\forall M_a \in \mathit{Models} : \exists n \in \mathbb{N} : n = \left|\widehat{\mathit{Traces}}^{M_a}\right|$) and a fair update strategy is applied (i.e. no approximation variable is permanently excluded from updates).

The presented iterative approach reaches a fixed point as soon as the following equation system is fulfilled.

$$\forall M_a \in \mathit{Models} : \widehat{\mathit{Approx}}^{M_a} = F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}}) \tag{4.33}$$

Any fixed point reached by our iteration starting from a maximally pessimistic initialization is guaranteed to coincide with the (unique) greatest fixed point of equation system (4.33). For a formal proof of this statement, we refer to page 249. As a consequence, the strategy according to which the approximation variables are updated (e.g. simultaneous update, cf. Figure 4.3) has no impact on the precision of a resulting fixed point.

In Chapter 7, we demonstrate how to instantiate this iterative approach for the calculation of *co-runner-sensitive* WCET bounds for multi-core processors with shared buses and event-driven bus arbitration. In this scenario, a compound consideration of the operation of all processor cores (i.e. an enumeration of all interleavings of accesses to the bus by the different processor cores) does not scale. Thus, we instantiate the presented iterative approach to safely overapproximate the results of a compound consideration. The instance relies on a property of *work-conserving* bus arbitration protocols: a processor core can only be blocked if a concurrent core is granted access. The abstract model of processor core C_i for which we want to calculate a WCET bound is pessimistically initialized to a set of abstract traces not making any assumption about the amount of shared-resource interference experienced by the concurrent processor cores (i.e. it can be arbitrarily high). The abstract model of each concurrent processor core C_j is pessimistically initialized to a set of abstract traces that might have an arbitrarily high execution time. Subsequently, the abstract trace sets of the different abstract models are updated in the following way:

- As a consequence of the work-conserving bus arbitration, core C_i must not be blocked for more cycles at the shared bus than the concurrent cores are granted access cycles.

- For each concurrent processor core C_j, we only have to consider the abstract traces describing traces that can happen during a program run on core C_i. Thus, we safely prune all abstract traces in the model of core C_j with a lower time bound exceeding the current WCET bound for core C_i.

This approach is safe. However, if there is not also a (most likely very pessimistic) upper bound on the number of blocked cycles that is independent of the programs executed on the concurrent processor cores, the initialization of the approach will already be a fixed point: If a program can be blocked arbitrarily long at the shared bus, we cannot upper bound its execution time. And while a program is executed for arbitrarily long, the concurrent cores can produce an unbounded amount of bus access cycles. This means a fixed point is reached.

Course of iteration:

iteration	$\widehat{Approx}^A$	$\widehat{Approx}^W$
init	$\{\widehat{t_a}\}$	$\{\widehat{t_w}\}$
1	$\{\widehat{t_a}\}$	$\{\widehat{t_w}, \widehat{t_x}\}$
2	$\{\widehat{t_a}, \widehat{t_b}\}$	$\{\widehat{t_w}, \widehat{t_x}\}$
3	$\{\widehat{t_a}, \widehat{t_b}\}$	$\{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}\}$
4	$\{\widehat{t_a}, \widehat{t_b}\}$	$\{\widehat{t_w}, \widehat{t_x}, \widehat{t_y}\}$

Figure 4.5.: Iterative overapproximation of the projections $\pi^A(\widehat{LessTraces})$ and $\pi^W(\widehat{LessTraces})$ presented in Figure 4.2. The iteration starts from an optimistic initialization. We apply the same update functions as in Figure 4.3 and a simultaneous update.

For some bus arbitration protocols (e.g. Round-Robin), the number of blocked cycles can also be bounded independently of the programs on the concurrent cores. However, if this is not the case, the presented iterative approach is mostly useless: for a priority-based bus arbitration, e.g., the number of blocked cycles can only be bounded independently of the concurrent cores for the core with the highest priority.

In order to overcome this problem and to potentially also improve the precision for other scenarios, we consider a potentially optimistic initialization in the next subsection.

4.2.4. Overapproximating the Projections by Starting from a Potentially Optimistic Initialization

Recall that every fixed point that our iterative approach can reach from a maximally pessimistic initialization provides an overapproximation of the projections of $\widehat{LessTraces}$ and coincides with the greatest fixed point of equation system (4.33). This raises the question whether any fixed point of equation system (4.33) provides an overapproximation of the projections of $\widehat{LessTraces}$. To answer this question, reconsider the example of Figure 4.3. This time, however, we optimistically initialize both approximation variables with the empty set. This results in an instantaneous fixed point as each of the update functions returns an empty set if its parameter is the empty set. Clearly, this fixed point does not provide an overapproximation of the projections of $\widehat{LessTraces}$ (cf. Figure 4.2). Thus, in general, not every fixed point of equation system (4.33) does provide an overapproximation of the projections.

Next, we consider the example of Figure 4.3 with a different optimistic initialization as shown in Figure 4.5. This time, every approximation variable is only initialized with a single abstract trace. We apply the same update functions and simultaneous update strategy as in Figure 4.3. For this optimistic initialization, a fixed point is reached after three iterations. The contents of the approximation variables at the fixed point do provide an overapproximation of the projections of $\widehat{LessTraces}$ (cf. Figure 4.2). This indicates that the choice of the initialization determines whether a fixed point reached from the initialization is a sound overapproximation of the projections. Intuitively, the update functions must be able to generate every member of every projection by a finite sequence of updates starting from the initialization in order to guarantee soundness. A formal soundness criterion will be presented later.

First, we formalize the setup of an overapproximation starting from a potentially optimistic initialization. We use $\widehat{Init^{M_a}}$ to refer to the initialization value of an approximation variable $\widehat{Approx^{M_a}}$. Thus, the choice of the vector $\overrightarrow{Init}$ determines the overall initialization.

$$\overrightarrow{Init} \equiv (\widehat{Init^{M_1}}, \dots, \widehat{Init^{M_m}}) \tag{4.34}$$

$$\forall M_a \in Models : \overrightarrow{Init_{\overline{M_a}}} \equiv (\widehat{Init^{M_1}}, \dots, \widehat{Init^{M_{a-1}}}, \widehat{Init^{M_{a+1}}}, \dots, \widehat{Init^{M_m}}) \tag{4.35}$$

These initialization values are used instead of the maximally pessimistic initialization (cf. equation (4.28)).

$$\forall M_a \in Models : \widehat{Approx^{M_a}} \leftarrow \widehat{Init^{M_a}} \tag{4.36}$$

Since we start from a potentially optimistic initialization, the idea is to add abstract traces to the approximation variables until a fixed point is reached. Thus, we shall choose the initialization in a way that any update function applied directly to the initialization does not reduce the abstract traces compared to the initialization.

$$\forall M_a \in Models : F^{M_a}(\overrightarrow{Init_{\overline{M_a}}}) \supseteq \widehat{Init^{M_a}} \tag{4.C4}$$

In combination with monotonicity criterion (4.C3), this guarantees that the updates of the approximation variables are *monotonically increasing*. This is expressed by hypothesis (4.H3). A formal proof of this hypothesis is analogous to the proof that the maximally pessimistic initialization makes the updates monotonically decreasing (cf. page 248) and, thus, omitted.

$$\forall M_a \in Models : F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) \supseteq \widehat{Approx^{M_a}} \tag{4.H3}$$

Any fixed point reached by our iteration starting from the chosen initialization is guaranteed to coincide with the (unique) least fixed point of the following equation system (4.37). Note that the initialization is incorporated in this equation system. For a formal proof of this statement, we refer to page 250. As a consequence, the strategy according to which the approximation variables are updated has no impact on the precision of a resulting fixed point.

$$\forall M_a \in Models : \widehat{Approx^{M_a}} = \widehat{Init^{M_a}} \cup F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) \tag{4.37}$$

A simultaneous update of some of the approximation variables is characterized by the non-empty subset of abstract models for which the variables are updated. The set *Update* contains all resulting possibilities to update one or more approximation variables simultaneously.

$$Update = \{upd \subseteq Models \mid upd \neq \emptyset\} \tag{4.38}$$

When an update is applied to a vector of approximation variables, it determines for which models the corresponding update functions are evaluated.

$$\begin{aligned} apply(\overrightarrow{Approx}, (upd)) &= (\widehat{Val^{M_1}}, \dots, \widehat{Val^{M_m}}) \\ \text{with } \forall M_a \in Models : \widehat{Val^{M_a}} &= \begin{cases} F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) & \text{, if } M_a \in upd \\ \widehat{Approx^{M_a}} & \text{, else} \end{cases} \end{aligned} \tag{4.39}$$

The definition of the set *UpdateSequ* of all possible (potentially empty) sequences of updates follows quite naturally.

$$UpdateSequ = \{()\} \cup \bigcup_{n \in \mathbb{N}_{\geq 1}} Update^n \tag{4.40}$$

An empty sequence of updates applied to a vector of approximation variables has no effect.

$$\mathit{apply}(\overrightarrow{\mathit{Approx}}, ()) = \overrightarrow{\mathit{Approx}} \tag{4.41}$$

The effect of applying a sequence of two or more updates to a vector of approximation variables is defined recursively.

$$\mathit{apply}(\overrightarrow{\mathit{Approx}}, (\mathit{upd}_1, \mathit{upd}_2, \ldots)) = \mathit{apply}(\mathit{apply}(\overrightarrow{\mathit{Approx}}, (\mathit{upd}_1)), (\mathit{upd}_2, \ldots)) \tag{4.42}$$

The aforementioned proof on page 250 also demonstrates that any sequence of updates applied to the initialization is guaranteed to not exceed the least fixed point of equation system (4.37).

$$\forall \mathit{updSequ} \in \mathit{UpdateSequ} : \mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ}) \subseteq_{\mathit{pairwise}} \overrightarrow{\mathbf{lfp}_{\text{equ.}(4.37)}} \tag{4.43}$$

Finally, we formalize a sufficient *soundness criterion* with respect to the initialization. It states that every abstract trace of every projection of $\widehat{\mathit{LessTraces}}$ shall be creatable by applying a sequence of updates to the initialization.

$$\begin{aligned}&\forall M_a \in \mathit{Models} : \forall \widehat{t^{M_a}} \in \pi^{M_a}(\widehat{\mathit{LessTraces}}) :\\ &\quad \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{M_a}} \in \pi^{M_a}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))\end{aligned} \tag{4.C5}$$

It follows from statement (4.43) and criterion (4.C5) that every component of the least fixed point of equation system (4.37) is an overapproximation of the corresponding projection of $\widehat{\mathit{LessTraces}}$.

$$\forall M_a \in \mathit{Models} : \pi^{M_a}(\widehat{\mathit{LessTraces}}) \subseteq \widehat{\mathbf{lfp}^{M_a}_{\text{equ.}(4.37)}} \tag{4.44}$$

Thus, any fixed point reached by the presented iterative approach—starting from an initialization fulfilling criteria (4.C4) and (4.C5)—is guaranteed to be an overapproximation of the projections of $\widehat{\mathit{LessTraces}}$.

Note that it is easy to manually verify that the optimistic example initialization of Figure 4.5 fulfills criterion (4.C5) as e.g. the update sequence $(\{W\}, \{A\})$ applied to the initialization is able to create all members of the projections of $\widehat{\mathit{LessTraces}}$. For more realistic scenarios, one typically needs an inductive proof that a given initialization fulfills criterion (4.C5).

Note that the iterative approach starting from a potentially optimistic initialization—as presented in this subsection—only guarantees an overapproximation of the projections of $\widehat{\mathit{LessTraces}}$ as soon as a fixed point is reached. The iterative approach starting from a maximally pessimistic initialization, in contrast, is an anytime algorithm. The latter might be helpful in case the precision of the approximation variables is already *good enough* (i.e. sufficient to prove the overall verification goal) after relatively few updates of the approximation variables. Thus, it depends on the actual use case which style of initialization is preferred.

4.2.5. Related Work

There is a worst-case response time analysis for multi-core processors with shared instruction caches [Liang et al., 2012] that is an instance of the overapproximation algorithm starting from a maximally pessimistic initialization as presented in Section 4.2.3. Initially, it calculates the WCET bounds assuming that each task can experience interference from all tasks executed on the concurrent processor cores. Subsequently, the algorithm calculates a lifetime interval per task based on a task dependence graph and the current WCET bounds of all tasks. The lifetime intervals are used to consider less potentially interfering concurrent tasks in the recalculation of the WCET bounds. These steps are repeated until the greatest fixed point is reached.

The verification of correctness properties of multi-threaded software (as e.g. mutual exclusion) is challenging as the precise consideration of all possible interleaving of accesses to the shared variables is in most cases practically intractable. This complexity can be significantly reduced by applying *thread-modular verification techniques* [Flanagan and Qadeer, 2003; Henzinger et al., 2003; Malkis et al., 2007; Gotsman et al., 2007; Miné, 2011, 2014; Monat and Miné, 2017]. The key idea of thread-modularity is to only consider the operation of one thread at a time—under a cumulative approximation of the possible interference that other threads can cause via the shared variables. Initially, it is assumed that no thread causes any interference. Subsequently, the interference created by a thread is recalculated under the assumption of the interference created by the other threads. This is repeated until a fixed point is reached. We believe that such thread-modular verification techniques can be derived in a structured and comprehensible way as instances of our iterative approach starting from a potentially optimistic initialization as presented in Section 4.2.4. It is, however, beyond the scope of this thesis to instantiate our iterative approach for thread-modular verification.

Note that the concept of thread-modularity is only a special case of using multiple abstract models that focus on different aspects of the concrete system. Thus, we believe that the presented iterative approaches can be used as foundation for the derivation of a wider range of compositional verification methods. In Chapter 7, we demonstrate their instantiation for the calculation of co-runner-sensitive WCET bounds for multi-core processors with shared buses and event-driven bus arbitration. The resulting instances of our iterative approaches can be seen as *processor-core-modular techniques* as they use one abstract model per processor core without resorting to an integrated consideration of the corresponding compound abstract model.

Further note that thread-modular model checking [Flanagan and Qadeer, 2003] has been shown [Malkis et al., 2006] to be an instance of abstract interpretation [Cousot and Cousot, 1977]. As noted before, we are confident that thread-modular model checking is also an instance of our iterative approach starting from a potentially optimistic initialization as presented in Section 4.2.4. It is, however, an open question whether there is a relation between our iterative approach and abstract interpretation. An interesting analogy between both approaches is that the least fixed point is only sound if a sufficient initialization is incorporated in the equation system. In abstract interpretation, this initialization typically has to provide a safe overapproximation of the program states at the start respectively the end of the program. In our approach, the initialization has to fulfill criteria (4.C4) and (4.C5) and is incorporated in equation system (4.37).

Chapter 5 A Hierarchy of Abstract Models

Sie ist ein Model und sie sieht gut aus.

(Das Model, Kraftwerk, 1978)

The worst-case execution time (WCET) of a program depends on its concrete traces at the micro-architectural level of the hardware platform it is executed on. This chapter defines the concrete traces of a state-based system and the corresponding system events in a generic way. Subsequently, it introduces a hierarchy of abstract models in order to consider the system's operation at a reduced degree of detail. Later chapters will make use of these ingredients in order to define the WCET of a program as well as algorithms that upper-bound its execution time.

5.1. Concrete Traces of a Concrete System

In this section, we specify a state-based system in a generic way. At each observable instant, the system has a state contained in a fixed set S of possible *system states.* When the system is started, it is in a state from a set of *initial system states.*

$$S_{init} \subseteq S \tag{5.1}$$

The state of the system changes per cycle tick following the relation *Cycle.* If and only if this relation holds for a pair of states (s_1, s_2), a cycle transition from s_1 to s_2 is possible.

$$Cycle \subseteq S \times S \tag{5.2}$$

We specify the set of all finite sequences of subsequent system states that are possible according to relation *Cycle* as follows.

$$Sequences = \{t : \mathbb{N}_{\leq n} \to S \mid n \in \mathbb{N} \wedge \forall x \in \mathbb{N}_{<n} : (t(x), t(x+1)) \in Cycle\} \tag{5.3}$$

We specify the set *Traces* of concrete traces as all members of *Sequences* starting from an initial system state. Intuitively, the set *Traces* contains all finite prefixes of all possible system executions.

$$Traces = \{t \in Sequences \mid t(0) \in S_{init}\} \tag{5.4}$$

We introduce the shorthand function *len* to obtain the length of a member of *Sequences.*

$$\forall n \in \mathbb{N} : \forall t \in Sequences \cap (\mathbb{N}_{\leq n} \to S) : len(t) = n \tag{5.5}$$

Events might occur during cycle transitions of the system. We define events in a way that it can be uniquely determined by the source and the target state of a cycle transition if the event happens. Thus, each event is uniquely described by the set of all cycle transitions in which it occurs. As a consequence, the set of all possible events can be defined as the power set of the relation *Cycle*.

$$\mathit{Events} = \mathcal{P}(\mathit{Cycle}) \tag{5.6}$$

As a consequence, *Cycle* is an example for an event of the system. It is the most general event as it happens during every cycle transition.

$$\mathit{Cycle} \in \mathit{Events} \tag{5.7}$$

Events that typically happen in the micro-architecture of a computer are e.g. cache misses or cycles blocked due to a DRAM refresh. In the context of WCET analysis for multi-core processors with shared buses, we are in particular interested in the events that a processor core C_i is blocked at the shared bus by the arbiter or granted access to it ($\mathit{Blocked}_{C_i}$, $\mathit{Granted}_{C_i}$, cf. Chapter 7). For now, we only consider generic events E.

$$E \in \mathit{Events} \tag{5.8}$$

Moreover, we use E as a shorthand to check whether event E happens at the cycle transition at a particular position of a member of *Sequences*.

$$E : \bigcup_{t \in \mathit{Sequences}} (\{t\} \times \mathbb{N}_{<\mathit{len}(t)}) \to \{0, 1\} \tag{5.9}$$

$$E(t, x) \equiv (t(x), t(x+1)) \in E \tag{5.10}$$

5.2. Approximation by Sequences of Abstract States

For modern computer systems, the set of possible system states is typically enormous. Thus, an explicit consideration of all possible concrete traces is practically infeasible. The state of the art in WCET analysis typically mitigates this problem by arguing about a fixed set $\widehat{S}$ of *abstract states*. Each abstract state describes a subset of the system states. This is expressed by a description function γ, which maps abstract states to sets of system states.

$$\gamma : \widehat{S} \to \mathcal{P}(S) \tag{5.11}$$

We select a subset of the abstract states as *initial abstract states*. It is chosen in a way that it describes all initial states of the concrete system under analysis.

$$\widehat{S_{\mathit{init}}} \subseteq \widehat{S} \tag{5.12}$$

$$\bigcup_{\widehat{s_i} \in \widehat{S_{\mathit{init}}}} \gamma(\widehat{s_i}) \supseteq S_{\mathit{init}} \tag{5.13}$$

The abstract state changes per cycle tick following the relation $\widehat{\mathit{Cycle}}$. If and only if this relation holds for a pair of abstract states $(\widehat{s_1}, \widehat{s_2})$, a cycle transition from $\widehat{s_1}$ to $\widehat{s_2}$ is possible.

$$\widehat{\mathit{Cycle}} \subseteq \widehat{S} \times \widehat{S} \tag{5.14}$$

The relation $\widehat{\mathit{Cycle}}$ shall describe at least all cycle transitions that are possible for concrete system states according to the relation *Cycle*.

$$\forall \widehat{s_1} \in \widehat{S} : \bigcup \{\gamma(\widehat{s_2}) \mid (\widehat{s_1}, \widehat{s_2}) \in \widehat{\mathit{Cycle}}\} \supseteq \bigcup_{s_1 \in \gamma(\widehat{s_1})} \{s_2 \mid (s_1, s_2) \in \mathit{Cycle}\} \tag{5.15}$$

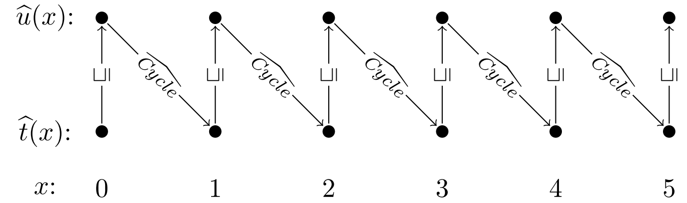


Figure 5.1.: Example for a $(\widehat{t},\widehat{u}) \in \widehat{\mathit{Sequences}}$ with $\mathit{len}((\widehat{t},\widehat{u})) = 5$.

An abstract state $\widehat{s_1}$ is called *at least as precise as* ($\sqsubseteq$) an abstract state $\widehat{s_2}$ if and only if the set of system states that $\widehat{s_1}$ describes does not exceed the set that $\widehat{s_2}$ describes.

$$\sqsubseteq\; \subseteq \widehat{S} \times \widehat{S} \tag{5.16}$$

$$\widehat{s_1} \sqsubseteq \widehat{s_2} \Leftrightarrow \gamma(\widehat{s_1}) \subseteq \gamma(\widehat{s_2}) \tag{5.17}$$

In order to enable an efficient implementation of the analysis, it is common to replace several similar abstract states by one abstract state that is at most as precise as each of the original abstract states. This process is typically referred to as *joining*. Similarly, one can replace a single abstract state $\widehat{s_1}$ by a less precise abstract state $\widehat{s_2}$ ($\widehat{s_1} \sqsubseteq \widehat{s_2}$ and $\widehat{s_1} \neq \widehat{s_2}$), which results in a *widening*.

Consider the following set $\widehat{\mathit{Sequences}}$ of all finite sequences of abstract states that can be created making use of $\sqsubseteq$ and $\widehat{\mathit{Cycle}}$ in all possible ways. Intuitively, each abstract state ($\widehat{t}(x)$) is replaced by an abstract state that can be less precise ($\widehat{u}(x)$) before the next abstract cycle transition. This principle is illustrated in Figure 5.1.

$$\begin{aligned} \widehat{\mathit{Sequences}} = \{(\widehat{t},\widehat{u}) \in (\mathbb{N}_{\leq n} \to \widehat{S})^2 \mid\ & n \in \mathbb{N}\ \wedge \\ & (\forall x \in \mathbb{N}_{\leq n} : \widehat{t}(x) \sqsubseteq \widehat{u}(x))\ \wedge \\ & \forall x \in \mathbb{N}_{<n} : (\widehat{u}(x), \widehat{t}(x+1)) \in \widehat{\mathit{Cycle}}\ \} \end{aligned} \tag{5.18}$$

We introduce the shorthand function *len* to obtain the length of a member of $\widehat{\mathit{Sequences}}$.

$$\forall n \in \mathbb{N} : \forall (\widehat{t},\widehat{u}) \in \widehat{\mathit{Sequences}} \cap (\mathbb{N}_{\leq n} \to \widehat{S})^2 : \mathit{len}((\widehat{t},\widehat{u})) = n \tag{5.19}$$

A *realistic analysis implementation*, however, will in most cases not make use of all possible ways in which an abstract state can be replaced by a (possibly different) abstract state according to $\sqsubseteq$. Thus, realistic analyses typically provide a subset of the members of $\widehat{\mathit{Sequences}}$.

$$\widehat{\mathit{Traces}} \subseteq \widehat{\mathit{Sequences}} \tag{5.20}$$

Any subset $\widehat{\mathit{Traces}}$ of $\widehat{\mathit{Sequences}}$ that fulfills the following two criteria is guaranteed to safely overapproximates the concrete traces. The first criterion states that every initial abstract state in $\widehat{S_{init}}$ must be the start abstract state of a prefix of length zero in $\widehat{\mathit{Traces}}$.

$$\forall \widehat{s_i} \in \widehat{S_{init}} : \exists (\widehat{t},\widehat{u}) \in \widehat{\mathit{Traces}} : \mathit{len}((\widehat{t},\widehat{u})) = 0 \wedge \widehat{t}(0) = \widehat{s_i} \tag{5.C1}$$

According to the second criterion, $\widehat{\mathit{Traces}}$ shall accumulate longer and longer prefixes by extending the existing prefixes by one cycle tick respectively. In this process, it must not forget any successor abstract state as dictated by $\widehat{\mathit{Cycle}}$.

$$\begin{aligned}
&\forall(\widehat{t_1},\widehat{u_1}) \in \widehat{\mathit{Traces}}:\\
&\quad\forall(\widehat{u_1}(\mathit{len}((\widehat{t_1},\widehat{u_1}))),\widehat{s_c}) \in \widehat{\mathit{Cycle}}:\\
&\quad\quad\exists(\widehat{t_2},\widehat{u_2}) \in \widehat{\mathit{Traces}}:\\
&\quad\quad\quad \mathit{len}((\widehat{t_2},\widehat{u_2})) = 1 + \mathit{len}((\widehat{t_1},\widehat{u_1})) \;\wedge\\
&\quad\quad\quad (\forall x \in \mathbb{N}_{\leq \mathit{len}((\widehat{t_1},\widehat{u_1}))} : \widehat{t_2}(x) = \widehat{t_1}(x) \wedge \widehat{u_2}(x) = \widehat{u_1}(x)) \;\wedge\\
&\quad\quad\quad \widehat{t_2}(\mathit{len}((\widehat{t_2},\widehat{u_2}))) = \widehat{s_c}
\end{aligned} \tag{5.C2}$$

Intuitively, these criteria are fulfilled by an abstract-interpretation-based program analysis that operates on a power-set-like domain of abstract states and uses a transfer function based on the relation $\widehat{\mathit{Cycle}}$ [Thesing, 2004]: it starts from a sound set of initial abstract states, may or may not make use of joining, but must never forget any successor abstract state according to relation $\widehat{\mathit{Cycle}}$. Thus, the result of a state-of-the-art micro-architectural analysis [Thesing, 2004] implicitly provides a set $\widehat{\mathit{Traces}}$ of prefixes that fulfills the two aforementioned criteria. For now, our formalism of prefixes of sequences of abstract states allows us to argue about the soundness of micro-architectural analysis without having to argue explicitly about program-analysis-specific particularities as e.g. control-flow graphs, context-sensitivity, fixed-points, or iteration. In Chapter 6, we will discuss and formalize some of these aspects. For a more in-depth discussion of these aspects, however, we refer to existing work [Cousot and Cousot, 1977; Nielson et al., 1999; Mauborgne and Rival, 2005].

We say that an abstract cycle transition $(\widehat{s_1},\widehat{s_2}) \in \widehat{\mathit{Cycle}}$ describes a concrete cycle transition $(s_1, s_2) \in \mathit{Cycle}$ if and only if $s_1 \in \gamma(\widehat{s_1})$ and $s_2 \in \gamma(\widehat{s_2})$. Thus, an abstract cycle transition potentially describes multiple concrete cycle transitions. An abstract trace from $\widehat{\mathit{Traces}}$ is a sequence of abstract cycle transitions (cf. equations (5.18) and (5.20)). Thus, each member of $\widehat{\mathit{Traces}}$ potentially describes multiple sequences of concrete cycle transitions.

Some of the sequences of concrete cycle transitions that a member of $\widehat{\mathit{Traces}}$ describes correspond to actual concrete traces from set *Traces*. The following relation *TraceDescrTrace* captures that a member of $\widehat{\mathit{Traces}}$ describes a concrete trace from set *Traces*.

$$\mathit{TraceDescrTrace} \subseteq \widehat{\mathit{Traces}} \times \mathit{Traces} \tag{5.21}$$

$$\begin{aligned}
&((\widehat{t},\widehat{u}), t) \in \mathit{TraceDescrTrace}\\
\Leftrightarrow\;& \mathit{len}((\widehat{t},\widehat{u})) = \mathit{len}(t) \wedge t(0) \in \gamma(\widehat{u}(0)) \wedge \forall x \in \mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq \mathit{len}(t)} : t(x) \in \gamma(\widehat{t}(x))
\end{aligned} \tag{5.22}$$

It follows that a set $\widehat{\mathit{Traces}}$ fulfilling criteria (5.C1) and (5.C2) describes each member of *Traces*. For a detailed proof of this statement, we refer to page 252.

$$\forall t \in \mathit{Traces} : \exists(\widehat{t},\widehat{u}) \in \widehat{\mathit{Traces}} : ((\widehat{t},\widehat{u}), t) \in \mathit{TraceDescrTrace} \tag{5.23}$$

A member of $\widehat{\mathit{Traces}}$, however, can also describe sequences of concrete cycle transitions that do not correspond to concrete traces from set *Traces*. This is the case if the source state of the first concrete cycle transition in a sequence is not an initial state of the concrete system (i.e. not $\in S_{init}$). Moreover, it is the case if the target state of a concrete cycle transition in a sequence does not coincide with the source state of the subsequent transition in the sequence. We call such sequences of concrete cycle transitions *spurious* as they cannot happen during any execution of the concrete system.

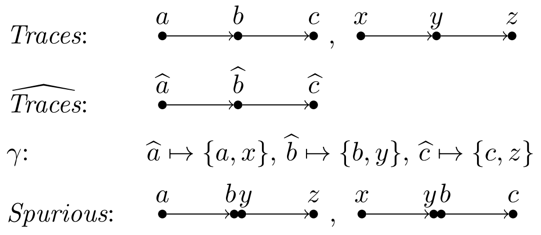


Figure 5.2.: Example for spurious sequences of cycle transitions

Figure 5.2 shows a sequence of abstract states that describes spurious sequences of concrete cycle transitions for a simple system with only two concrete traces. The figure only contains complete sequences of length three for both system and abstract system states. The shorter prefixes have been omitted for simplicity. Furthermore, we assume the relation $\sqsubseteq$ to coincide with the identity relation. Thus, we do not explicitly represent joining/widening in this figure (i.e. $\forall(\widehat{t}, \widehat{u}) \in \widehat{\mathit{Traces}} : \forall n \in \mathbb{N}_{\leq \mathit{len}(\widehat{t})} : \widehat{t}(n) = \widehat{u}(n)$). The abstract cycle transition $(\widehat{a}, \widehat{b})$ in Figure 5.2 describes the concrete cycle transitions (a, b) and (x, y). In the same way, $(\widehat{b}, \widehat{c})$ describes (b, c) and (y, z). As a consequence, the two possible concrete traces from *Traces* are soundly described by the single sequence of abstract states in $\widehat{\mathit{Traces}}$. The sequence of abstract states, however, also describes two spurious sequences of concrete cycle transitions. They cannot appear on the concrete system since the target state of one cycle transition does not coincide with the source state of the next transition (e.g. $b \neq y$).

We formally define the set of all spurious sequences of concrete cycle transitions as follows.

$$\begin{aligned} \mathit{Spurious} = \{(t, u) \in (\mathbb{N}_{<n} \to S) \times ((\mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n}) \to S) \\ \mid n \in \mathbb{N} \wedge [\forall x \in \mathbb{N}_{<n} : (t(x), u(x+1)) \in \mathit{Cycle}] \\ \wedge [t(0) \notin S_{\mathit{init}} \vee \exists x \in (\mathbb{N}_{\geq 1} \cap \mathbb{N}_{<n}) : t(x) \neq u(x)]\} \end{aligned} \tag{5.24}$$

The length of a spurious sequence of cycle transitions is defined analogously to the length of concrete traces or sequences of abstract states.

$$\forall n \in \mathbb{N} : \forall (t, u) \in (\mathit{Spurious} \cap ((\mathbb{N}_{<n} \to S) \times ((\mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n}) \to S))) : \mathit{len}((t, u)) = n \tag{5.25}$$

We use E as a shorthand to check whether event E happens at a particular position in a spurious sequence of cycle transitions of *Spurious*.

$$E : \bigcup_{(t,u) \in \mathit{Spurious}} (\{(t, u)\} \times \mathbb{N}_{<\mathit{len}((t,u))}) \to \{0, 1\} \tag{5.26}$$

$$E((t, u), x) \equiv (t(x), u(x+1)) \in E \tag{5.27}$$

The set $\widehat{\mathit{Traces}}$ can also describe some spurious traces (i.e. traces which do not belong to the concrete traces). We formally define the description of such spurious traces by the relation *TraceDescrSpurious*.

$$\mathit{TraceDescrSpurious} \subseteq \widehat{\mathit{Traces}} \times \mathit{Spurious} \tag{5.28}$$

$$\begin{aligned} & ((\widehat{t}, \widehat{u}), (t, u)) \in \mathit{TraceDescrSpurious} \\ \Leftrightarrow \; & \mathit{len}((\widehat{t}, \widehat{u})) = \mathit{len}((t, u)) \\ & \wedge [\forall x \in \mathbb{N}_{<\mathit{len}(t)} : t(x) \in \gamma(\widehat{u}(x))] \\ & \wedge \forall x \in (\mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq \mathit{len}(t)}) : u(x) \in \gamma(\widehat{t}(x)) \end{aligned} \tag{5.29}$$

Finally, we define the function γ_{trace} that maps a sequence of abstract states from $\widehat{Traces}$ to the concrete traces and/or spurious sequences of cycle transitions that it describes.

$$\gamma_{trace} : \widehat{Traces} \rightarrow \mathcal{P}(Traces \cup Spurious) \tag{5.30}$$

$$\begin{aligned} \gamma_{trace}((\widehat{t}, \widehat{u})) = \{&t \in Traces \mid ((\widehat{t}, \widehat{u}), t) \in TraceDescrTrace\} \cup \\ \{&(t, u) \in Spurious \mid ((\widehat{t}, \widehat{u}), (t, u)) \in TraceDescrSpurious\} \end{aligned} \tag{5.31}$$

It follows from equations (5.22), (5.29), and (5.31) that any prefix of a sequence of abstract states can only describe prefixes of concrete and/or spurious traces that have the same length as the describing prefix.

$$\forall \widehat{t} \in \widehat{Traces} : \forall t \in \gamma_{trace}(\widehat{t}) : len(\widehat{t}) = len(t) \tag{5.32}$$

Note that—according to statement (5.23)—$(\widehat{Traces}, \gamma_{trace})$ is an overapproximation of *Traces*.

$$\bigcup_{\widehat{t} \in \widehat{Traces}} \gamma_{trace}(\widehat{t}) \supseteq Traces \tag{5.33}$$

Thus, $(\widehat{Traces}, \gamma_{trace})$ is an abstract model of *Traces* as defined in Chapter 4. As a consequence, we can apply property lifting to it.

The system properties that we are going to consider argue about the events occurring in the concrete traces. Consequently, the lifted versions of these system properties will also have to argue about events based on the knowledge that the abstract model offers. To this end, we introduce a concept of events on cycle transitions of abstract states. It is defined analogous to the events on cycle transitions of system states.

$$\widehat{Events} = \mathcal{P}(\widehat{Cycle}) \tag{5.34}$$

$$\widehat{E} \in \widehat{Events} \tag{5.35}$$

We use $\widehat{E}$ as a shorthand to check whether event $\widehat{E}$ happens at the cycle transition at a particular position in a sequence of abstract states of $\widehat{Sequences}$.

$$\widehat{E} : \bigcup_{(\widehat{t}, \widehat{u}) \in \widehat{Sequences}} (\{(\widehat{t}, \widehat{u})\} \times \mathbb{N}_{<len((\widehat{t}, \widehat{u}))}) \rightarrow \{0, 1\} \tag{5.36}$$

$$\widehat{E}((\widehat{t}, \widehat{u}), x) \equiv (\widehat{u}(x), \widehat{t}(x+1)) \in \widehat{E} \tag{5.37}$$

However, the events on cycle transitions of abstract states must be related to the corresponding events on cycle transitions of system states. A cycle transition on abstract states might not always have precise knowledge about whether a particular event happens in all cycle transitions of system states that it describes. Thus, we introduce the concept of *event bounds*. Each event E on cycle transitions of system states shall be bounded from above by a *may-event* $\widehat{E^{UB}}$ on cycle transitions of abstract states. Intuitively, as soon as an abstract cycle transition describes a transition on system states for which an event E appears, the corresponding may-event $\widehat{E^{UB}}$ must appear for the abstract cycle transition.

$$\forall (\widehat{s_1}, \widehat{s_2}) \in \widehat{Cycle} : E \cap (\gamma(\widehat{s_1}) \times \gamma(\widehat{s_2})) \neq \emptyset \Rightarrow (\widehat{s_1}, \widehat{s_2}) \in \widehat{E^{UB}} \tag{5.38}$$

Analogously, each event E on cycle transitions of system states shall be bounded from below by a *must-event* $\widehat{E^{LB}}$ on cycle transitions of abstract states. Intuitively, as soon as an abstract cycle transition describes a transition on system states for which an event E does not appear, the corresponding must-event $\widehat{E^{LB}}$ must not appear for the abstract cycle transition.

$$\forall (\widehat{s_1}, \widehat{s_2}) \in \widehat{Cycle} : (Cycle \setminus E) \cap (\gamma(\widehat{s_1}) \times \gamma(\widehat{s_2})) \neq \emptyset \Rightarrow (\widehat{s_1}, \widehat{s_2}) \notin \widehat{E^{LB}} \tag{5.39}$$

As the event *Cycle* appears—by definition—for every cycle transition on system states, it is safe to assume that the corresponding must-event appears for every cycle transition on abstract states.

$$\widehat{Cycle^{LB}} = \widehat{Cycle} = \widehat{Cycle^{UB}} \tag{5.40}$$

Moreover, note that it is safe for every event E to assume that the corresponding may-event $\widehat{E^{UB}}$ coincides with $\widehat{Cycle}$ and the corresponding must-event $\widehat{E^{LB}}$ coincides with $\emptyset$. Intuitively, this means that the event E may always occur according to the event bounds, but the event bounds can never guarantee that it occurs.

We can use the event bounds to argue about the events at an arbitrary position of a concrete or spurious trace described by a sequence of abstract states. For a detailed proof of this statement, we refer to page 254.

$$\begin{gathered}\forall \widehat{t} \in \widehat{Traces} : \forall t \in \gamma_{trace}(\widehat{t}) : \forall x \in \mathbb{N}_{<len(\widehat{t})} : \\ \widehat{E^{LB}}(\widehat{t}, x) \leq E(t, x) \leq \widehat{E^{UB}}(\widehat{t}, x)\end{gathered} \tag{5.41}$$

It follows that we can bound the number of times an event E happens during a subset of all positions of a concrete or spurious trace described by a sequence of abstract states.

$$\begin{gathered}\forall \widehat{t} \in \widehat{Traces} : \forall t \in \gamma_{trace}(\widehat{t}) : \forall X \subseteq \mathbb{N}_{<len(\widehat{t})} : \\ \sum_{x \in X} \widehat{E^{LB}}(\widehat{t}, x) \leq \sum_{x \in X} E(t, x) \leq \sum_{x \in X} \widehat{E^{UB}}(\widehat{t}, x)\end{gathered} \tag{5.42}$$

We exploit the event bounds when lifting system properties to the approximation level of sequences of abstract states. Concrete properties are Boolean predicates on concrete and spurious traces (i.e. $P_k : \mathit{Traces} \cup \mathit{Spurious} \to \{0, 1\}$). In the same way, properties on sequences of abstract states are logical formulae that map sequences of abstract states to truth values (i.e. $\widehat{P_k} : \widehat{Traces} \to \{0, 1\}$). Based on the results of Chapter 4, we know that a lifted version $\widehat{P_k}$ of a system property P_k can safely be used to detect infeasible abstract traces if it fulfills criterion (4.C1):

$$\forall \widehat{t} \in \widehat{Traces} : [\, \exists t \in \gamma_{trace}(\widehat{t}) : P_k(t)\,] \Rightarrow \widehat{P_k}(\widehat{t})$$

For simple system properties, a lifted version satisfying criterion (4.C1) can easily be obtained based on intuition. This is demonstrated by the example property P_{xmpl} and its lifted version in Figure 5.3. Essentially, property P_{xmpl} states that a trace t must not exhibit more occurrences of event E_1 than of event E_2. We can safely lift this property to a sequence of abstract states $\widehat{t}$ by replacing the left-hand side of the original inequation by a lower bound on the number of occurrences of event E_1 in $\widehat{t}$. In the same way, the right-hand side of the original inequation is replaced by an upper bound on the number of occurrences of event E_2 in $\widehat{t}$. Thus, if $\widehat{t}$ describes only a single trace in which the number of occurrences of event E_1 does not exceed the number of occurrences of event E_2, the lifted property $\widehat{P_{xmpl}}$ is guaranteed to hold for $\widehat{t}$. If, in contrast, $\widehat{P_{xmpl}}$ does not hold for a sequence $\widehat{t}$, we know that $\widehat{t}$ can only describe traces in which the number of occurrences of event E_1 exceeds the number of occurrences of event E_2. If we moreover know that P_{xmpl} is a system property, $\widehat{t}$ cannot describe any concrete traces of the concrete system and, thus, is infeasible.

Real-world system properties are typically more complex than the simple example in Figure 5.3. To avoid the cumbersome task of manually lifting every considered system property to sequences of abstract states in a way that fulfills criterion (4.C1), we provide a set of generic lifting rules that can be applied in a more or less mechanical way. Table 5.1 contains the lifting rules in the

$$P_{xmpl}(t) \Leftrightarrow \sum_{x\in\mathbb{N}_{<len(t)}} E_1(t,x) \leq \sum_{x\in\mathbb{N}_{<len(t)}} E_2(t,x)$$

$$\widehat{P_{xmpl}}(\widehat{t}) \Leftrightarrow \sum_{x\in\mathbb{N}_{<len(\widehat{t})}} \widehat{E_1^{LB}}(\widehat{t},x) \leq \sum_{x\in\mathbb{N}_{<len(\widehat{t})}} \widehat{E_2^{UB}}(\widehat{t},x)$$

Figure 5.3.: Example for a simple system property P_{xmpl} and a lifted version of it that fulfills criterion (4.C1)

form of a function ***lift*** for the most common logical constructs of the properties we consider. The logical constructs of properties are listed in the first column. The second column shows how to safely lift them to sequences of abstract states based on safely lifted versions of their children. The lifting rules make use of an auxiliary function ***flip*** that recursively flips the event bounds in a property on sequences of abstract states. It is defined in Table 5.2. For a formal proof of the soundness of the lifting rules, we refer to page 255.

We conclude this section by showing how the example property of Figure 5.3 is lifted to sequences of abstract states by applying the rules of Table 5.1 and Table 5.2:

$$\begin{aligned}
& \boldsymbol{lift}(P_{xmpl}(t)) \\
\underset{\text{Figure 5.3}}{\Leftrightarrow}\ & \boldsymbol{lift}\left(\textstyle\sum_{x\in\mathbb{N}_{<len(t)}} E_1(t,x) \leq \sum_{x\in\mathbb{N}_{<len(t)}} E_2(t,x)\right) \\
\underset{\text{(LR7)}}{\Leftrightarrow}\ & \textstyle\sum_{x\in\mathbb{N}_{<len(t)}} \boldsymbol{flip}(\boldsymbol{lift}(E_1(t,x))) \leq \sum_{x\in\mathbb{N}_{<len(t)}} \boldsymbol{lift}(E_2(t,x)) \\
\underset{\text{(LR1)}}{\Leftrightarrow}\ & \textstyle\sum_{x\in\mathbb{N}_{<len(t)}} \boldsymbol{flip}(\widehat{E_1^{UB}}(\widehat{t},x)) \leq \sum_{x\in\mathbb{N}_{<len(t)}} \widehat{E_2^{UB}}(\widehat{t},x) \\
\underset{\text{(LR9)}}{\Leftrightarrow}\ & \textstyle\sum_{x\in\mathbb{N}_{<len(t)}} \widehat{E_1^{LB}}(\widehat{t},x) \leq \sum_{x\in\mathbb{N}_{<len(t)}} \widehat{E_2^{UB}}(\widehat{t},x) \\
\underset{\text{(5.32)}}{\Leftrightarrow}\ & \textstyle\sum_{x\in\mathbb{N}_{<len(\widehat{t})}} \widehat{E_1^{LB}}(\widehat{t},x) \leq \sum_{x\in\mathbb{N}_{<len(\widehat{t})}} \widehat{E_2^{UB}}(\widehat{t},x) \\
\underset{\text{Figure 5.3}}{\Leftrightarrow}\ & \widehat{P_{xmpl}}(\widehat{t})
\end{aligned}$$

$P(t, \vec{x})$	$\boldsymbol{lift}(P(t, \vec{x}))$	
$E(t, x_i)$	$\widehat{E^{UB}}(\widehat{t}, x_i)$	(LR1)
$[\ P_1(t, \vec{x}) \wedge P_2(t, \vec{x})\ ]$	$[\ \boldsymbol{lift}(P_1(t, \vec{x})) \wedge \boldsymbol{lift}(P_2(t, \vec{x}))\ ]$	(LR2)
$[\ P_1(t, \vec{x}) \vee P_2(t, \vec{x})\ ]$	$[\ \boldsymbol{lift}(P_1(t, \vec{x})) \vee \boldsymbol{lift}(P_2(t, \vec{x}))\ ]$	(LR3)
$[\ \forall x \in X : P_1(t, \vec{x}, x)\ ]$	$[\ \forall x \in X : \boldsymbol{lift}(P_1(t, \vec{x}, x))\ ]$	(LR4)
$[\ \exists x \in X : P_1(t, \vec{x}, x)\ ]$	$[\ \exists x \in X : \boldsymbol{lift}(P_1(t, \vec{x}, x))\ ]$	(LR5)
$\neg P_1(t, \vec{x})$	$\neg \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x})))$	(LR6)
$[\ a_1 \cdot \sum_{x \in X} P_1(t, \vec{x}, x) + b_1$ $\lesssim a_2 \cdot \sum_{y \in Y} P_2(t, \vec{x}, y) + b_2\ ]$	$[\ a_1 \cdot \sum_{x \in X} \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}, x))) + b_1$ $\lesssim a_2 \cdot \sum_{y \in Y} \boldsymbol{lift}(P_2(t, \vec{x}, y)) + b_2\ ]$	(LR7)
$[\ P_1(t, \vec{x}, x) \Rightarrow P_2(t, \vec{x}, x)\ ]$	$[\ \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}, x)))$ $\Rightarrow \boldsymbol{lift}(P_2(t, \vec{x}, x))\ ]$	(LR8)
$E \in \mathit{Events} \qquad X, Y \subseteq \mathbb{N} \qquad x, y$ fresh variables w.r.t. $\vec{x}$ $a_1, a_2 \in \mathbb{R}_{\geq 0} \qquad b_1, b_2 \in \mathbb{R} \qquad \lesssim\ \in \{<, \leq\}$		

Table 5.1.: Rules for lifting properties from concrete traces to sequences of abstract states. The auxiliary function ***flip*** used in the lifting rules is defined in Table 5.2.

$\widehat{P}(\widehat{t}, \vec{x})$	$\boldsymbol{flip}(\widehat{P}(\widehat{t}, \vec{x}))$	
$\widehat{E^{UB}}(\widehat{t}, x_i)$	$\widehat{E^{LB}}(\widehat{t}, x_i)$	(LR9)
$\widehat{E^{LB}}(\widehat{t}, x_i)$	$\widehat{E^{UB}}(\widehat{t}, x_i)$	(LR10)
$[\ \widehat{P_1}(\widehat{t}, \vec{x}) \wedge \widehat{P_2}(\widehat{t}, \vec{x})\ ]$	$[\ \boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x})) \wedge \boldsymbol{flip}(\widehat{P_2}(\widehat{t}, \vec{x}))\ ]$	(LR11)
$[\ \widehat{P_1}(\widehat{t}, \vec{x}) \vee \widehat{P_2}(\widehat{t}, \vec{x})\ ]$	$[\ \boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x})) \vee \boldsymbol{flip}(\widehat{P_2}(\widehat{t}, \vec{x}))\ ]$	(LR12)
$[\ \forall x \in X : \widehat{P_1}(\widehat{t}, \vec{x}, x)\ ]$	$[\ \forall x \in X : \boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x}, x))\ ]$	(LR13)
$[\ \exists x \in X : \widehat{P_1}(\widehat{t}, \vec{x}, x)\ ]$	$[\ \exists x \in X : \boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x}, x))\ ]$	(LR14)
$\neg\widehat{P_1}(\widehat{t}, \vec{x})$	$\neg\boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x}))$	(LR15)
$[\ a_1 \cdot \sum_{x \in X} \widehat{P_1}(\widehat{t}, \vec{x}, x) + b_1$ $\lesssim a_2 \cdot \sum_{y \in Y} \widehat{P_2}(\widehat{t}, \vec{x}, y) + b_2\ ]$	$[\ a_1 \cdot \sum_{x \in X} \boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x}, x)) + b_1$ $\lesssim a_2 \cdot \sum_{y \in Y} \boldsymbol{flip}(\widehat{P_2}(\widehat{t}, \vec{x}, y)) + b_2\ ]$	(LR16)
$[\ \widehat{P_1}(\widehat{t}, \vec{x}, x) \Rightarrow \widehat{P_2}(\widehat{t}, \vec{x}, x)\ ]$	$[\ \boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x}, x)) \Rightarrow \boldsymbol{flip}(\widehat{P_2}(\widehat{t}, \vec{x}, x))\ ]$	(LR17)
$\boldsymbol{flip}(\widehat{P_1}(\widehat{t}, \vec{x}, x))$	$\widehat{P_1}(\widehat{t}, \vec{x}, x)$	(LR18)
$E \in$ *Events* $\quad X, Y \subseteq \mathbb{N} \quad x, y$ fresh variables w.r.t. $\vec{x}$ $a_1, a_2 \in \mathbb{R}_{\geq 0} \quad b_1, b_2 \in \mathbb{R} \quad \lesssim\ \in \{<, \leq\}$		

Table 5.2.: Auxiliary function ***flip*** used when lifting properties from concrete traces to sequences of abstract states. It recursively flips the directions of the event bounds in a property on sequences of abstract states.

5.3. Approximation by Paths through a Graph

There are also typically too many different sequences of abstract states to exhaustively account for all of them. As a next level of approximation, the sequences of abstract states are approximated by paths through a graph. A graph is typically defined on a set of *nodes*. A subset of the nodes is chosen as *start nodes*.

$$\mathit{Nodes}_{\mathit{start}} \subseteq \mathit{Nodes} \tag{5.43}$$

Similarly, a subset of the nodes is chosen as *end nodes*.

$$\mathit{Nodes}_{\mathit{end}} \subseteq \mathit{Nodes} \tag{5.44}$$

The *edges* are defined as a subset of all possible pairs of nodes. As a consequence, we obtain a *directed graph*.

$$\mathit{Edges} \subseteq \mathit{Nodes} \times \mathit{Nodes} \tag{5.45}$$

A path between any pair of nodes is referred to as *subpath*.

$$\widehat{\mathit{SubPaths}} = \{\widehat{p} : \mathbb{N}_{\leq n} \to \mathit{Nodes} \mid n \in \mathbb{N} \land \forall x \in \mathbb{N}_{<n} : (\widehat{p}(x), \widehat{p}(x+1)) \in \mathit{Edges}\} \tag{5.46}$$

We define the *length* of a subpath as the number of edges it consists of.

$$\forall n \in \mathbb{N} : \forall \widehat{p} \in \widehat{\mathit{SubPaths}} \cap (\mathbb{N}_{\leq n} \to \mathit{Nodes}) : \mathit{len}(\widehat{p}) = n \tag{5.47}$$

The subpaths that begin with a start node are referred to as *relaxed paths*.

$$\widehat{\mathit{RelPaths}} = \{\widehat{p} \in \widehat{\mathit{SubPaths}} \mid \widehat{p}(0) \in \mathit{Nodes}_{\mathit{start}}\} \tag{5.48}$$

The actual *paths through the graph* are those relaxed paths that additionally end in an end node. They are given by the set $\widehat{\mathit{Paths}}$.

$$\widehat{\mathit{Paths}} = \{\widehat{p} \in \widehat{\mathit{RelPaths}} \mid \widehat{p}(\mathit{len}(\widehat{p})) \in \mathit{Nodes}_{\mathit{end}}\} \tag{5.49}$$

Note that, in general, not every node in the graph is guaranteed to reach one of the end nodes. Thus, there can be relaxed paths that are not a prefix of one of the members of $\widehat{\mathit{Paths}}$.

For now, our focus is on the paths from a start node to an end node. However, the notions of subpaths and relaxed paths are required in Chapter 6 of this thesis. Thus, we define most of the following formal machinery directly on subpaths in order to reuse it later.

In order to argue about the graph in a convenient way, we define the sets of *predecessors* and *successors* per graph node.

$$\mathit{pred}, \mathit{succ} : \mathit{Nodes} \to \mathcal{P}(\mathit{Nodes}) \tag{5.50}$$

$$\mathit{pred}(\mathit{node}) = \{\mathit{node}' \in \mathit{Nodes} \mid (\mathit{node}', \mathit{node}) \in \mathit{Edges}\} \tag{5.51}$$

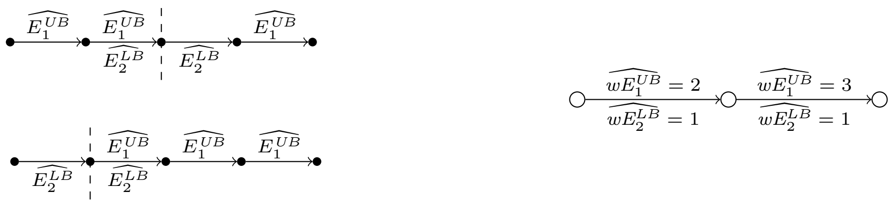
$$\mathit{succ}(\mathit{node}) = \{\mathit{node}' \in \mathit{Nodes} \mid (\mathit{node}, \mathit{node}') \in \mathit{Edges}\} \tag{5.52}$$

Moreover, we define the sets of *in-edges* and *out-edges* per graph node.

$$\mathit{inEdges}, \mathit{outEdges} : \mathit{Nodes} \to \mathcal{P}(\mathit{Edges}) \tag{5.53}$$

$$\mathit{inEdges}(\mathit{node}) = \{(\mathit{node}_{\mathit{src}}, \mathit{node}_{\mathit{trgt}}) \in \mathit{Edges} \mid \mathit{node}_{\mathit{trgt}} = \mathit{node}\} \tag{5.54}$$

$$\mathit{outEdges}(\mathit{node}) = \{(\mathit{node}_{\mathit{src}}, \mathit{node}_{\mathit{trgt}}) \in \mathit{Edges} \mid \mathit{node}_{\mathit{src}} = \mathit{node}\} \tag{5.55}$$



(a) Two sequences of abstract cycle transitions.

(b) A path describing the sequences.

Figure 5.4.: Example of a path that describes sequences of abstract cycle transitions.

We define the set of *start edges* as the out-edges of the start nodes.

$$Edges_{start} = \bigcup_{node_{start} \in Nodes_{start}} outEdges(node_{start}) \tag{5.56}$$

Analogously, we define the set of *end edges* as the in-edges of the end nodes.

$$Edges_{end} = \bigcup_{node_{end} \in Nodes_{end}} inEdges(node_{end}) \tag{5.57}$$

In a weighted graph, an *edge weight* $\widehat{w}$ is a function that maps the edges to weight values.

$$\widehat{Weights} = (Edges \to \mathbb{N}) \tag{5.58}$$

$$\widehat{w} \in \widehat{Weights} \tag{5.59}$$

Moreover, we use $\widehat{w}$ as a shorthand to the value of the function $\widehat{w}$ for an edge at a particular position of a path.

$$\widehat{w} : \bigcup_{\widehat{p} \in \widehat{SubPaths}} (\{\widehat{p}\} \times \mathbb{N}_{<len(\widehat{p})}) \to \mathbb{N} \tag{5.60}$$

$$\widehat{w}(\widehat{p}, x) \equiv \widehat{w}((\widehat{p}(x), \widehat{p}(x+1))) \tag{5.61}$$

For each event E, there shall be two corresponding edge weights.

$$\widehat{wE^{UB}},\ \widehat{wE^{LB}} \in \widehat{Weights} \tag{5.62}$$

An edge describes a sequence of abstract cycle transitions if, for all events E, the sum over the may events $\widehat{E^{UB}}$ along the sequence is upper-bounded by the weight $\widehat{wE^{UB}}$ of the edge and the sum over the must events $\widehat{E^{LB}}$ along the sequence is lower-bounded by the weight $\widehat{wE^{LB}}$ of the edge. Thus, $edg \in Edges$ describes sequences of abstract cycle transitions in which the event E occurs at most $\widehat{wE^{UB}}(edg)$ times and at least $\widehat{wE^{LB}}(edg)$ times. A path consisting of x edges describes a sequence of abstract cycle transitions if we can split up the sequence into x sub-sequences in such a way that each sub-sequence is described by the corresponding edge in the path.

Figure 5.4 demonstrates this principle. Figure 5.4a shows two sequences of four abstract cycle transitions respectively. Figure 5.4b shows a path. Each sequence of Figure 5.4a can be split—as indicated by the dashed line—into two sub-sequences in a way that the first sub-sequence is described by the first edge of the path and the second sub-sequence is described by the second edge of the path. Thus, each sequence is described by the path.

In order to model the different ways in which a sequence of abstract cycle transitions can be split into a given number of sub-sequences, we introduce the formal concept of a partitioning. A partitioning from the set $Partitionings(v, w)$ splits the range of the first w natural numbers (0 to

$part \in Partitionings(4,7)$

$part(0) = 2$	$part(1) = 0$	$part(2) = 3$	$part(3) = 2$
$\{0,1\}$	$\{\}$	$\{2,3,4\}$	$\{5,6\}$
$from(part,0) = 0$	$from(part,1) = 2$	$from(part,2) = 2$	$from(part,3) = 5$
$to(part,0) = 1$	$to(part,1) = 1$	$to(part,2) = 4$	$to(part,3) = 6$

Figure 5.5.: Example for a partitioning that groups the first 7 natural numbers into 4 partitions of subsequent numbers

$w - 1$) into v partitions of subsequent numbers (partition 0 to partition $v - 1$). A partitioning is encoded as a function *part* that assigns each partition its respective size.

$$Partitionings : \mathbb{N} \times \mathbb{N} \to \mathcal{P}(\{part : \mathbb{N}_{<n} \to \mathbb{N} \mid n \in \mathbb{N}\}) \tag{5.63}$$

$$Partitionings(v,w) = \{part : \mathbb{N}_{<v} \to \mathbb{N} \mid \sum_{x \in \mathbb{N}_{<v}} part(x) = w\} \tag{5.64}$$

Figure 5.5 shows an example for a partitioning that groups the first 7 natural numbers into 4 partitions of subsequent numbers. Partition 0 is assigned a size of two. Thus, it contains the first two natural numbers ($\{0,1\}$). Partition 1, in contrast, is assigned a size of zero. Consequently, it is empty ($\{\}$). The contents of the remaining partitions are defined analogously.

In order to easily argue about the content of partition x in a partitioning *part*, we use the helper functions *from* and *to*. Partition x contains all natural numbers that are not smaller than $from(part,x)$ and do not exceed $to(part,x)$. The helper functions *from* and *to* are formally defined by the following equations.

$$from, to : \bigcup_{a \in \mathbb{N}} \bigcup_{b \in \mathbb{N}} \bigcup_{part \in Partitionings(a,b)} (\{part\} \times \mathbb{N}_{<a}) \to \mathbb{N} \tag{5.65}$$

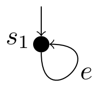
$$from(part,x) = \sum_{y \in \mathbb{N}_{<x}} part(y) \tag{5.66}$$

$$to(part,x) = from(part,x) + part(x) - 1 \tag{5.67}$$

We formally define that a path describes a particular sequence of abstract states if and only if there is a partitioning of the corresponding sequence of abstract cycle transitions such that the event bounds per partition x are safely bounded from above and below by the corresponding weights of the edge at position x of the path. This is expressed by relation *PathDescrTrace*.

$$PathDescrTrace \subseteq \widehat{SubPaths} \times \widehat{Traces} \tag{5.68}$$

$$\begin{aligned}
&(\widehat{p}, \widehat{t}) \in PathDescrTrace \\
&\Leftrightarrow \exists part \in Partitionings(len(\widehat{p}), len(\widehat{t})) : \\
&\quad \forall E \in Events : \\
&\qquad \forall x \in \mathbb{N}_{<len(\widehat{p})} : \\
&\qquad\quad \sum_{from(part,x) \le i \le to(part,x)} \widehat{E^{UB}}(\widehat{t}, i) \le \widehat{wE^{UB}}(\widehat{p}, x) \wedge \\
&\qquad\quad \sum_{from(part,x) \le i \le to(part,x)} \widehat{E^{LB}}(\widehat{t}, i) \ge \widehat{wE^{LB}}(\widehat{p}, x)
\end{aligned} \tag{5.69}$$



$$\widehat{wCycle}^{UB}(e) = \widehat{wCycle}^{LB}(e) = 1$$
$$\forall E \in \mathit{Events} \setminus \{\mathit{Cycle}\} : \widehat{wE}^{UB}(e) = 1 \wedge \widehat{wE}^{LB}(e) = 0$$

Figure 5.6.: Example for a trivial graph that soundly approximates essentially any system.

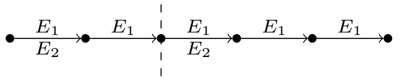


Figure 5.7.: Example for a *spurious sequence of events* described by the path of Figure 5.4b. It is spurious with respect to the sequences of abstract cycle transitions presented in Figure 5.4a (i.e. not safely bounded by one of them).

We require the following criterion to hold for the graphs that we consider.

$$\forall \widehat{t} \in \widehat{\mathit{Traces}} : \exists \widehat{p} \in \widehat{\mathit{Paths}} : (\widehat{p}, \widehat{t}) \in \mathit{PathDescrTrace} \tag{5.C3}$$

Criterion (5.C3) implies that the paths through the graph describe all sequences of abstract states.

$$\widehat{\mathit{Traces}} = \{\widehat{t} \in \widehat{\mathit{Traces}} \mid \widehat{p} \in \widehat{\mathit{Paths}} \wedge (\widehat{p}, \widehat{t}) \in \mathit{PathDescrTrace}\} \tag{5.70}$$

As an example, consider the graph in Figure 5.6. It only consists of a single node and a single edge forming a self-loop. The edge corresponds to a single cycle transition of the system. All events except the event *Cycle* are modeled in a very pessimistic way: at each instant, an event might or might not occur. Intuitively, the graph soundly describes the concrete traces of essentially every possible system. However, since it lost all information about the events happening, it will hardly be of any use during validation of a system's operation.

Note that an edge approximates the sequences of abstract cycle transitions in a cumulative way. It e.g. approximates away the order in which the respective may- and must-events of the described sequences occur. As a consequence, a path through a graph can describe more than only the sequences of abstract states contained in $\widehat{\mathit{Traces}}$. It can additionally describe *spurious sequences of events* that are not covered by the sequences of abstract states it describes. Figure 5.7 presents an example of a sequence of events described by the path of Figure 5.4b. The sequence is spurious with respect to the sequences of abstract cycle transitions presented in Figure 5.4a as it is not safely bounded by one of them. The spurious sequence e.g. features five occurrences of event E_1 while each sequence of Figure 5.4a allows at most three.

In the following, we formally define the sequences of events that are spurious with respect to $\widehat{\mathit{Traces}}$. None of them is safely bounded by the may- and must-events of a member of $\widehat{\mathit{Traces}}$.

$$\begin{aligned}\widehat{\mathit{SpuriousTraces}} = \{\widehat{st} : \mathbb{N}_{<n} \to \mathcal{P}(\mathit{Events}) \\ \mid n \in \mathbb{N} \wedge \\ \forall \widehat{t} \in \widehat{\mathit{Traces}} : \\ [\mathit{len}(\widehat{t}) = n] \Rightarrow \\ [\exists x \in \mathbb{N}_{<n} : \exists E \in \mathit{Events} : \\ (E \in \widehat{st}(x) \wedge \neg \widehat{E^{UB}}(\widehat{t}, x)) \vee \\ (E \notin \widehat{st}(x) \wedge \widehat{E^{LB}}(\widehat{t}, x))]\}\end{aligned} \tag{5.71}$$

The length of a member of $\widehat{SpuriousTraces}$ is defined analogously to the length of a sequence of abstract states.

$$\forall n \in \mathbb{N} : \forall \widehat{st} \in (\widehat{SpuriousTraces} \cap (\mathbb{N}_{<n} \to \mathcal{P}(Events))) : len(\widehat{st}) = n \tag{5.72}$$

We also define the event bounds for the members of $\widehat{SpuriousTraces}$ in a way that they provide the same interface as the event bounds for sequences of abstract states.

$$\widehat{E^{UB}},\ \widehat{E^{LB}} : \bigcup_{\widehat{st} \in \widehat{SpuriousTraces}} (\{\widehat{st}\} \times \mathbb{N}_{<len(\widehat{st})}) \to \{0,1\} \tag{5.73}$$

$$\widehat{E^{UB}}(\widehat{st}, x) \equiv \widehat{E^{LB}}(\widehat{st}, x) \equiv E \in \widehat{st}(x) \tag{5.74}$$

In this way, we can use the properties on sequences of abstract states also for the members of $\widehat{SpuriousTraces}$.

Moreover, this allows us to specify the description relation *PathDescrSpuriousTrace* from paths to members of $\widehat{SpuriousTraces}$ analogously to the relation *PathDescrTrace*.

$$PathDescrSpuriousTrace \subseteq \widehat{SubPaths} \times \widehat{SpuriousTraces} \tag{5.75}$$

$$\begin{aligned}
&(\widehat{p}, \widehat{st}) \in PathDescrSpuriousTrace \\
\Leftrightarrow &\exists part \in Partitionings(len(\widehat{p}), len(\widehat{st})) : \\
&\quad \forall E \in Events : \\
&\quad\quad \forall x \in \mathbb{N}_{<len(\widehat{p})} : \\
&\quad\quad\quad \sum_{from(part,x) \le i \le to(part,x)} \widehat{E^{UB}}(\widehat{st}, i) \le \widehat{wE^{UB}}(\widehat{p}, x) \wedge \\
&\quad\quad\quad \sum_{from(part,x) \le i \le to(part,x)} \widehat{E^{LB}}(\widehat{st}, i) \ge \widehat{wE^{LB}}(\widehat{p}, x)
\end{aligned} \tag{5.76}$$

Finally, we define the function γ_{path} that maps paths from $\widehat{SubPaths}$ to the members of $\widehat{Traces}$ and/or $\widehat{SpuriousTraces}$ that it describes.

$$\gamma_{path} : \widehat{SubPaths} \to \mathcal{P}(\widehat{Traces} \cup \widehat{SpuriousTraces}) \tag{5.77}$$

$$\begin{aligned}
\gamma_{path}(\widehat{p}) = &\{\widehat{t} \in \widehat{Traces} \mid (\widehat{p}, \widehat{t}) \in PathDescrTrace\} \cup \\
&\{\widehat{st} \in \widehat{SpuriousTraces} \mid (\widehat{p}, \widehat{st}) \in PathDescrSpuriousTrace\}
\end{aligned} \tag{5.78}$$

Note that—according to statement (5.70)—$(\widehat{Paths}, \gamma_{path})$ is an overapproximation of $\widehat{Traces}$.

$$\bigcup_{\widehat{p} \in \widehat{Paths}} \gamma_{path}(\widehat{p}) \supseteq \widehat{Traces} \tag{5.79}$$

Consequently, $(\widehat{Paths}, \gamma_{path})$ is also an overapproximation of the subset of $\widehat{Traces}$ for which all the lifted system properties hold.

$$\bigcup_{\widehat{p} \in \widehat{Paths}} \gamma_{path}(\widehat{p}) \supseteq \{\widehat{t} \in \widehat{Traces} \mid \forall P_k \in Prop : \widehat{P_k}(\widehat{t})\} \tag{5.80}$$

We know from Section 5.2 that this subset of $\widehat{Traces}$ provides an overapproximation of the set *Traces* of concrete traces. At the same time, however, we can see this subset as a system on its own for which the system properties $\widehat{P_k}$ hold. Thus, we can once more apply property

$\widehat{P}(\widehat{t})$	$\boldsymbol{lift}(\widehat{P}(\widehat{t}))$
$\forall z \in \mathbb{N}_{\leq len(\widehat{t})} :$ $a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1$ $\lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2$	$\forall z \in \mathbb{N}_{\leq len(\widehat{p})} :$ $a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{wE_1^{LB}}(\widehat{p}, x) + b_1$ $\lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{wE_2^{UB}}(\widehat{p}, x) + b_2$
$E_1, E_2 \in \mathit{Events} \qquad a_1, a_2 \in \mathbb{R}_{\geq 0} \qquad b_1, b_2 \in \mathbb{R} \qquad \lesssim \in \{<, \leq\}$	

(LR19)

Table 5.3.: Rule for lifting a common type of property from sequences of abstract states to paths.

lifting to further lift the properties $\widehat{P_k}$ to the approximation level of the paths through the graph. The properties lifted to the paths are annotated with the superscript *path* in order to easily distinguish them from the properties lifted to sequences of abstract states. Consequently, we rephrase soundness criterion (4.C1) in the following way for this step of further lifting an already lifted property.

$$\forall \widehat{p} \in \widehat{SubPaths} : [\, \exists \widehat{t} \in \gamma_{path}(\widehat{p}) : \widehat{P_k}(\widehat{t}) \,] \Rightarrow \widehat{P_k^{path}}(\widehat{p}) \tag{5.C4}$$

Lifting system properties from sequences of abstract states to paths is—to the best of our knowledge—not as straight forward as lifting system properties from concrete traces to sequences of abstract states. Intuitively, the lifting is more challenging as a graph edge approximates away the order of the event bounds of the different sub-sequences of abstract states. It is an open question whether there is a set of lifting rules that is similarly generic as the rules presented in the previous section. The answer to this open question, however, is beyond the scope of this thesis. Nonetheless, Table 5.3 provides a single lifting rule from sequences of abstract states to paths. A soundness proof of this lifting rule can be found on page 262.

5.4. Approximation by Implicit Path Enumeration

The previous section formalizes the approximation by paths through a graph. Depending on the size of the graph representation, it may be practically infeasible to explicitly argue about each path induced by the graph. As a potential remedy, it is common to abstract away from the order in which the edges appear in a path through the graph. This approach is referred to as *implicit path enumeration (technique)*—or short IPET—and has been successfully used in WCET analysis for a long time [Li and Malik, 1995; Stein, 2010]. In this section, we formalize implicit path enumeration as another level of approximation in our formal framework. Moreover, we show how system properties lifted to paths can be further lifted to this level of approximation in a safe way.

The implicit path enumeration—in its general form that we discuss here—only argues about the number of times each edge occurs in a potential path and which edges start and end such a potential path. For our formal definition, we exploit some observations about the paths:

- an edge can only be the start edge of a path if it is contained in the path
- an edge can only be the end edge of a path if it is contained in the path
- in general, a path $\widehat{p}$ may be empty (i.e. it contains no edge at all, $len(\widehat{p}) = 0$)
- a non-empty path has exactly one start edge and exactly one end edge

- a non-empty path starts from a start edge
- a non-empty path ends in an end edge
- if the sets of start nodes and end nodes of the graph are disjoint, there cannot be an empty path through the graph (i.e. all paths must have a start edge)
- the number of times an out-edge of a particular node is taken in a path must coincide with the number of times an in-edge of the node is taken
  - exception: one out-edge more if the path starts in an out-edge of the node
  - exception: one in-edge more if the path ends in an in-edge of the node

This leads to the following formal definition of the set of abstract traces that we consider in implicit path enumeration. We informally refer to the members of this set as implicit paths.

$$\begin{aligned}\widehat{\mathit{Implicit}} = \{ & (\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd}) \in (\mathit{Edges} \to \mathbb{N}) \times (\mathit{Edges} \to \{0,1\})^2 \mid \\ & [\ \forall e \in \mathit{Edges} : \mathit{isStart}(e) \leq \mathit{timesTaken}(e)\ ] \wedge \\ & [\ \forall e \in \mathit{Edges} : \mathit{isEnd}(e) \leq \mathit{timesTaken}(e)\ ] \wedge \\ & \sum_{e \in \mathit{Edges}} \mathit{isStart}(e) \leq 1 \wedge \\ & \sum_{e \in \mathit{Edges}} \mathit{isEnd}(e) \leq 1 \wedge \\ & \sum_{e \in \mathit{Edges}} \mathit{isStart}(e) = \sum_{e \in \mathit{Edges}} \mathit{isEnd}(e) \wedge \\ & \sum_{e \in (\mathit{Edges} \setminus \mathit{Edges}_{start})} \mathit{isStart}(e) = 0 \wedge \\ & \sum_{e \in (\mathit{Edges} \setminus \mathit{Edges}_{end})} \mathit{isEnd}(e) = 0 \wedge \\ & [\ \mathit{Nodes}_{start} \cap \mathit{Nodes}_{end} = \emptyset \Rightarrow \sum_{e \in \mathit{Edges}} \mathit{isStart}(e) = 1\ ] \wedge \\ & [\ \forall \mathit{node} \in \mathit{Nodes} : \\ & \qquad \sum_{e_{in} \in \mathit{inEdges}(\mathit{node})} [\mathit{timesTaken}(e_{in}) - \mathit{isEnd}(e_{in})] \\ & \qquad = \sum_{e_{out} \in \mathit{outEdges}(\mathit{node})} [\mathit{timesTaken}(e_{out}) - \mathit{isStart}(e_{out})]\ ] \\ \} & \end{aligned} \tag{5.81}$$

Intuitively, an implicit path describes an actual path if its function *timesTaken* returns for every edge the number of times the edge is contained in the actual path and its functions *isStart* and *isEnd* only return 1 for the first respectively last edge in the actual path. This is formally expressed by the relation *ImplicitDescrPath*.

$$\mathit{ImplicitDescrPath} \subseteq \widehat{\mathit{Implicit}} \times \widehat{\mathit{Paths}} \tag{5.82}$$

$$\begin{aligned} & ((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd}), \widehat{p}) \in \mathit{ImplicitDescrPath} \\ \Leftrightarrow & \forall e \in \mathit{Edges} : \\ & \quad \mathit{timesTaken}(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \wedge \\ & \quad \mathit{isStart}(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \wedge \\ & \quad \mathit{isEnd}(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \end{aligned} \tag{5.83}$$

It follows that—with respect to this description relation—the set $\widehat{Implicit}$ describes all paths contained in the set $\widehat{Paths}$. For a detailed proof of this statement, we refer to page 268.

$$\widehat{Paths} = \{\widehat{p} \in \widehat{Paths} \mid \widehat{i} \in \widehat{Implicit} \wedge (\widehat{i}, \widehat{p}) \in ImplicitDescrPath\} \tag{5.84}$$

An implicit path, however, might also describe sequences of edges that do not correspond to actual paths in the graph. We refer to them as *spurious paths* and define them as follows.

$$\begin{aligned} \widehat{SpuriousPaths} = \{\widehat{p} : \mathbb{N}_{<n} \to Edges \mid n \in \mathbb{N}_{>0} \wedge \\ [\,(\neg\exists nd \in Nodes_{start} : \widehat{p}(0) \in outEdges(nd)) \vee \\ (\neg\exists nd \in Nodes_{end} : \widehat{p}(n-1) \in inEdges(nd)) \vee \\ \exists x \in \mathbb{N}_{<n-1} : \\ \exists (nd_1, nd_2), (nd_3, nd_4) \in Edges : \\ (nd_1, nd_2) = \widehat{p}(x) \wedge (nd_3, nd_4) = \widehat{p}(x+1) \wedge nd_2 \neq nd_3 \,]\} \end{aligned} \tag{5.85}$$

We define the *length* of a spurious path as the number of edges it consists of.

$$\forall n \in \mathbb{N} : \forall \widehat{p} \in \widehat{SpuriousPaths} \cap (\mathbb{N}_{<n} \to Edges) : len(\widehat{p}) = n \tag{5.86}$$

For each edge weight $\widehat{w} \in \widehat{Weights}$, we use $\widehat{w}$ as a shorthand to the value of the function $\widehat{w}$ for an edge at a particular position of a spurious path.

$$\widehat{w} : \bigcup_{\widehat{p} \in \widehat{SpuriousPaths}} (\{\widehat{p}\} \times \mathbb{N}_{<len(\widehat{p})}) \to \mathbb{N} \tag{5.87}$$

$$\widehat{w}(\widehat{p}, x) \equiv \widehat{w}(\widehat{p}(x)) \tag{5.88}$$

The description relation between implicit paths and spurious paths is formally defined in the following way.

$$ImplicitDescrSpuriousPath \subseteq \widehat{Implicit} \times \widehat{SpuriousPaths} \tag{5.89}$$

$$\begin{aligned} &((timesTaken, isStart, isEnd), \widehat{p}) \in ImplicitDescrSpuriousPath \\ \Leftrightarrow &\forall e \in Edges : \\ &\quad timesTaken(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid \widehat{p}(x) = e\}\right| \wedge \\ &\quad isStart(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge \widehat{p}(x) = e\}\right| \wedge \\ &\quad isEnd(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge \widehat{p}(x) = e\}\right| \end{aligned} \tag{5.90}$$

Finally, we define the function γ_{impli} that maps implicit paths from $\widehat{Implicit}$ to the members of $\widehat{Paths}$ and/or $\widehat{SpuriousPaths}$ that it describes.

$$\gamma_{impli} : \widehat{Implicit} \to \mathcal{P}(\widehat{Paths} \cup \widehat{SpuriousPaths}) \tag{5.91}$$

$$\begin{aligned} \gamma_{impli}(\widehat{i}) = \{\widehat{p} \in \widehat{Paths} \mid (\widehat{i}, \widehat{p}) \in ImplicitDescrPath\} \cup \\ \{\widehat{p} \in \widehat{SpuriousPaths} \mid (\widehat{i}, \widehat{p}) \in ImplicitDescrSpuriousPath\} \end{aligned} \tag{5.92}$$

Note that—according to statement (5.84)—$(\widehat{Implicit}, \gamma_{impli})$ is an overapproximation of $\widehat{Paths}$.

$$\bigcup_{\widehat{i} \in \widehat{Implicit}} \gamma_{impli}(\widehat{i}) \supseteq \widehat{Paths} \tag{5.93}$$

Consequently, $(\widehat{Implicit}, \gamma_{impli})$ is also an overapproximation of the subset of $\widehat{Paths}$ for which all the lifted system properties hold.

$$\bigcup_{\widehat{i} \in \widehat{Implicit}} \gamma_{impli}(\widehat{i}) \supseteq \{\widehat{p} \in \widehat{Paths} \mid \forall P_k \in Prop : \widehat{P_k^{path}}(\widehat{p})\} \tag{5.94}$$

Thus, we can once more apply property lifting to further lift the properties $\widehat{P_k^{path}}$ to the approximation level of the implicit path enumeration. The properties lifted to the implicit path enumeration are annotated with the superscript *impli*. Consequently, we rephrase soundness criterion (4.C1) in the following way for this step of further lifting an already lifted property.

$$\forall \widehat{i} \in \widehat{Implicit} : [\, \exists \widehat{p} \in \gamma_{impli}(\widehat{i}) : \widehat{P_k^{path}}(\widehat{p}) \,] \Rightarrow \widehat{P_k^{impli}}(\widehat{i}) \tag{5.C5}$$

Note that there are implicit paths which do not describe any actual path through the graph (i.e. only spurious paths). A smart choice of the lifted properties used during implicit path enumeration, however, guarantees that such implicit paths are pruned [Puschner and Schedl, 1997]. We made similar observations while conducting experiments with our analysis prototype.

In order to effectively lift properties from paths to implicit paths, we need to argue about the sums over different edge weights along paths described by an implicit path. Intuitively, an implicit path has exact knowledge about the number of times each graph edge is contained in all paths described by it. Thus, we can exactly specify the sum over all occurrences of a particular edge weight along a described path.

$$\begin{gathered}\forall \widehat{w} \in \widehat{Weights} : \forall \widehat{i} \in \widehat{Implicit} : \forall \widehat{p} \in \gamma_{impli}(\widehat{i}) : \\ \sum_{e \in Edges} timesTaken(e) \cdot \widehat{w}(e) = \sum_{x \in \mathbb{N}_{< len(\widehat{p})}} \widehat{w}(\widehat{p}, x)\end{gathered} \tag{5.95}$$

Moreover, it has exact knowledge about the first edge and the last edge of all paths described by it.

$$\begin{gathered}\forall \widehat{w} \in \widehat{Weights} : \forall \widehat{i} \in \widehat{Implicit} : \forall \widehat{p} \in \gamma_{impli}(\widehat{i}) : \\ \sum_{e \in Edges} isStart(e) \cdot \widehat{w}(e) = \sum_{x \in (\mathbb{N}_{< len(\widehat{p})} \cap \{0\})} \widehat{w}(\widehat{p}, x)\end{gathered} \tag{5.96}$$

$$\begin{gathered}\forall \widehat{w} \in \widehat{Weights} : \forall \widehat{i} \in \widehat{Implicit} : \forall \widehat{p} \in \gamma_{impli}(\widehat{i}) : \\ \sum_{e \in Edges} isEnd(e) \cdot \widehat{w}(e) = \sum_{x \in (\mathbb{N}_{< len(\widehat{p})} \cap \{len(\widehat{p})-1\})} \widehat{w}(\widehat{p}, x)\end{gathered} \tag{5.97}$$

The formal proofs of these statements can be found from page 275 onwards.

Table 5.4 presents a rule for lifting a common type of system property from paths to implicit paths. For a formal proof that the rule fulfills criterion (5.C5), we refer to page 276.

$\widehat{P^{path}}(\widehat{p})$	$\boldsymbol{\mathit{lift}}(\widehat{P^{path}}(\widehat{p}))$
$\forall z \in \mathbb{N}_{\leq len(\widehat{p})}: a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_1}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_2}(\widehat{p}, x) + b_2$	$a_1 \cdot \sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w_1}(e) + b_1 \lesssim a_2 \cdot \sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w_2}(e) + b_2$
$\widehat{w_1}, \widehat{w_2} \in \widehat{Weights} \qquad a_1, a_2 \in \mathbb{R}_{\geq 0} \qquad b_1, b_2 \in \mathbb{R} \qquad \lesssim \in \{<, \leq\}$	

(LR20)

Table 5.4.: Rule for lifting a common type of property from paths to implicit paths.

5.5. Relevance and Related Work

This chapter formally defines the concrete traces of a state-based system in a generic way. The concrete traces determine which events occur during the execution of the system. Based on this, we introduce a hierarchy of abstract models that argue about the concrete traces—and, thus, also about the system events—at a reduced degree of detail. Each higher level of approximation is formally shown to be an abstract model—as defined in Chapter 4—of the underlying level. Thus, each higher level of approximation describes at least the traces of its underlying level. A schematic overview of the hierarchy of abstract models is shown in Figure 5.8.

The typical WCET analysis workflow (i.e. a micro-architectural analysis and a subsequent implicit path enumeration) can be seen as operating on the three levels of approximation that we formalized:

1. A micro-architectural analysis [Thesing, 2004] operates on an approximation by sequences of abstract states. The sequences are implicitly given by the fixed point of an abstract interpretation [Cousot and Cousot, 1977].

2. In a post-processing step, the results of a micro-architectural analysis are represented as a directed, weighted graph [Matthies, 2006; Stein, 2010]. The length of a (finite) longest path (w.r.t. the edge weight upper bounding the number of clock cycles) through the graph is a WCET bound [Matthies, 2006].

3. Finally, an implicit path enumeration is performed in order to safely approximate the length of a longest path (w.r.t. the edge weight upper bounding the number of clock cycles) through the graph [Li and Malik, 1995; Puschner and Schedl, 1997; Theiling, 2002; Stein, 2010].

Approximation is typically applied in order to *gain efficiency* when verifying the correct operation of a concrete system. Intuitively, an abstract model typically features significantly fewer abstract traces than the number of concrete traces because each abstract trace potentially describes multiple concrete traces.

The typical drawback of the use of approximation is a *reduced precision.* As a consequence, a verification goal (e.g. that the WCET of a program does not exceed a deadline) might be impossible to show based on a given abstract model (i.e. the WCET bound obtained from the abstract model exceeds the deadline) although it holds for the concrete system (i.e. the actual WCET would not have exceeded the deadline, but its exact calculation is not tractable with current techniques).

The loss in precision stems from an abstract model typically describing more than the concrete traces of the concrete system. The abstract model of each of the three levels of approximation shown in Figure 5.8 describes *spurious traces* that do not belong to the traces of the level below.

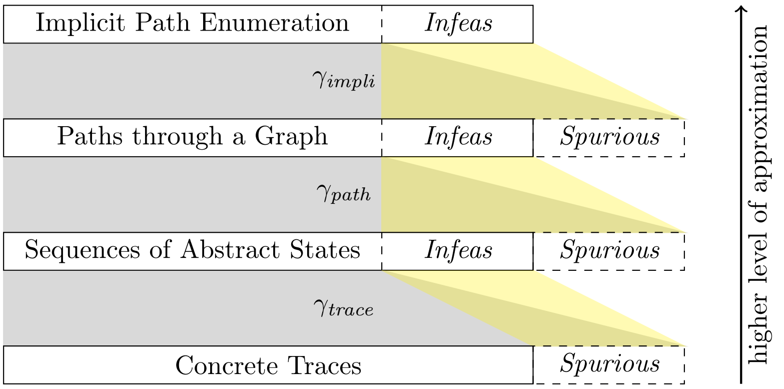


Figure 5.8.: Schematic overview of the hierarchy of abstract models:
▨ Each *feasible* abstract trace describes some feasible traces of the level below.
🟨 An *infeasible* abstract trace does not describe any feasible trace of the level below.

At each level of approximation, there is a subset of so-called *infeasible abstract traces* which only describe spurious and/or infeasible traces of the underlying level (cf. parts labeled with *Infeas* in Figure 5.8). Infeasible abstract traces can safely be pruned. Thus, it is desirable to efficiently detect infeasible abstract traces. It is clearly not efficient to explicitly consider the actually described traces for each abstract trace in order to decide if it is infeasible. The principle of *property lifting* allows us to efficiently detect some of the infeasible abstract traces. Note that, in general, we cannot expect to detect all infeasible abstract traces with the help of lifted system properties: their use is *sound* but *not necessarily complete* (cf. example at the end of Section 3.1).

The idea of using additional knowledge about a concrete system in order to improve an abstract model of the concrete system is not entirely new. In WCET analysis, it is common to encode such knowledge as additional constraints in an implicit path enumeration based on integer linear programming (ILP). Early publications about implicit path enumeration [Li and Malik, 1995] already propose to encode *loop bounds* in this way. Later publications [Engblom and Ermedahl, 2000; Raymond, 2014] propose to encode more general *flow facts* as linear constraints in order to exclude a wider range of infeasible paths from the calculation of the WCET bound. Moreover, it has been proposed to encode bounds on the numbers of cache hits and misses as ILP constraints in order to reduce the WCET bound [Li et al., 1995, 1996]. Similarly, knowledge about the persistence of a particular cache block in a particular region of a program is also encoded as ILP constraints [Stein, 2010; Cullmann, 2013]. The principle of cache persistence has later been generalized to a path-sensitive cache analysis [Nagar and Srikant, 2015], which uses ILP constraints to encode that certain cache misses can only occur if particular paths triggering them are executed. It has also been proposed to use ILP constraints for bounding the number of write-backs in a cache [Blaß et al., 2017]. Stein recognizes in his dissertation [Stein, 2010] that the inclusion of cache-persistence constraints is a special case of a more general class of cumulative constraints that can improve the precision of a WCET bound. The formalism used in his dissertation, however, is not strong enough to argue about the soundness of incorporating such additional constraints. To the best of our knowledge, we are first to formalize the three common levels of approximation used in WCET analysis in a unified framework and to provide general criteria for soundly lifting arbitrary properties of the concrete system up the hierarchy of abstract models. Thus, the aforementioned approaches can be seen as special instances of the more general framework of property lifting.

Note that the concept of an *infeasible abstract trace* is a generalization of the concept of an infeasible path as it is classically used in literature. The term infeasible path classically refers to a sequence of basic blocks that is possible while traversing a control-flow graph (CFG) of a particular program, but not when actually executing the program [Hedley and Hennell, 1985]. Intuitively, the same is covered by the term infeasible abstract trace as defined by us: an abstract trace is infeasible if it only describes traces corresponding to sequences of basic blocks that are not possible during an actual execution of the program. Moreover, however, an abstract trace is also infeasible if it e.g. only describes traces that experience an amount of shared-resource interference that is higher than possible on the concrete system. The system properties used for the detection of infeasible paths classically only argue about which basic blocks are executed how often (and potentially in which order they are executed). The property lifting framework, in contrast, supports the lifting of properties that argue about arbitrary system events. The inclusion of cache constraints in implicit path enumeration [Li et al., 1995, 1996; Stein, 2010; Cullmann, 2013; Nagar and Srikant, 2015; Blaß et al., 2017] is an example for instances of our framework that go beyond the detection of infeasible paths in the classical sense. Be aware that an infeasible abstract trace in an approximation by paths through a graph (cf. Section 5.3) is intuitively also referred to as an infeasible path.

Implicit path enumeration has been successfully used for the verification of timing-critical systems for a long time [Li and Malik, 1995; Puschner and Schedl, 1997; Theiling, 2002; Stein, 2010]. The soundness arguments of the early variants [Li and Malik, 1995; Puschner and Schedl, 1997], however, are mostly based on intuition. Later publications [Theiling, 2002; Stein, 2010] do not argue about the soundness at all. In the same way, the inclusion of additional constraints has also mostly been based on intuition [Li et al., 1995, 1996; Engblom and Ermedahl, 2000; Stein, 2010; Cullmann, 2013; Nagar and Srikant, 2015; Blaß et al., 2017]. The only studies going beyond intuition are limited to system properties arguing about the control flow (i.e. flow facts) [Blazy et al., 2013; Maroneze et al., 2014; Raymond, 2014; Mussot and Sotin, 2015]. We close this gap with our work: From page 268 onwards, we formally prove that the baseline abstract model of implicit path enumeration argues about all paths through the graph of the underlying level of approximation. Moreover, we provide formal criteria for safely lifting arbitrary system properties to the level of approximation of implicit path enumeration (4.C1, 5.C4, 5.C5).

Further note that we formalized a very general form of implicit path enumeration that is independent of the implementing technology. In particular, it does not necessarily have to be implemented as ILP problem. We might for example use lifted system properties that cannot be expressed as a set of linear constraints. As a consequence, we do not take into account the problems that might arise due to numerical instabilities [Higham, 2002] in the actual implementing technology. Moreover, according to our view, the loop bounds are not a part of the baseline abstract model of implicit path enumeration. They are later incorporated as part of the set of lifted system properties in order to detect infeasible implicit paths. The original version of implicit path enumeration [Li and Malik, 1995], in contrast, was presented as an ILP formulation that already included the loop-bounding constraints.

Finally, note that—so far—we only presented the hierarchy of abstract models that WCET analysis typically operates on. The following chapter will explain how this hierarchy is used to bound—from above or from below—the number of times that a particular event might occur during any execution run of a particular program. The calculation of a WCET bound can be seen as a special case of such a bound calculation.

Chapter 6 Calculation of Event Bounds

We have approximately 35 seconds...
or less

(T-800, Terminator Genisys, 2015)

The worst-case execution time (WCET) of a program is specified by means of micro-architectural events. Intuitively, it is the maximal amount of clock cycle events spent for executing the program between a program start event and the subsequent program end event. The WCET of a program is only defined if the program is guaranteed to not diverge—which is undecidable in general due to the halting problem.

The WCET is a special case of a more general class of exact event bounds that can be specified over the execution runs of a program. One can e.g. specify the maximal amount of write accesses to a particular memory range during any execution run of a considered program analogously by replacing the event *Cycle* by a correspondingly restricted event. Similarly, one can specify the minimal amount of occurrences of a particular event during any execution run of a considered program. In general, it depends on the termination of a program whether these actual maxima and minima are defined, i.e. whether there are finite maximal and minimal values.

In this chapter, we formally specify the maximal and minimal amount of occurrences of a particular event during any execution run of a considered program. Subsequently, we show how to safely bound these extremal values—the maximum from above and the minimum from below—with the help of the hierarchy of abstract models defined in Chapter 5.

WCET analysis typically only considers all paths from the program start to the program end. For the calculation of general event bounds (i.e. the considered event does no coincide with the clock cycle event), this approach is sound for programs that are guaranteed to terminate. For programs that can diverge, in contrast, the classical approach of only considering paths from the program start to the program end can lead to an underestimation (overestimation) of the actual maximum (minimum). Section 3.3 demonstrates these soundness issues at an intuitive level. In this chapter, we formalize the calculation of event bounds. Moreover, we present a slight extension of the classical approach of only considering paths from the program start to the program end. Our extended event bound calculation workflow relies on the notion of feedback node sets [Karp, 1972]. It is shown to also be sound for the calculation of general event bounds for programs that can diverge.

6.1. Program Execution Runs

Let *InstrMemAddr* be the space of available addresses in the physical instruction memory of the system. Each system state s shall be uniquely mapped to the address of the instruction that will be executed during the next clock cycle starting from s on processor core C_i. This mapping is given by the function *instrAddr*.

$$\mathit{instrAddr} : S \times \mathit{Cores} \rightarrow \mathit{InstrMemAddr} \tag{6.1}$$

For processors that only process one instruction at a time per processor core, the mapping is canonical. Modern processor cores, however, typically perform a pipelined execution in order to exploit instruction-level parallelism [Hennessy and Patterson, 2011]. Thus, *instrAddr* has to logically assign each system state to only one of the instructions that are in the pipeline of the considered processor core. In this respect, we follow the convention used in the dissertation of Stephan Thesing [Thesing, 2004]. It assigns each system state to the first instruction in the pipeline that will leave the pipeline again. In combination with in-order execution, it is straight forward to identify this particular instruction in the pipeline (cf. example on page 109 of Thesing's dissertation). Implementations of out-of-order execution typically rely on an in-order retirement of the instructions [Hennessy and Patterson, 2011] and, thus, also support an easy determination of the instruction that will leave the pipeline next.

We define an event that occurs if and only if a particular *instruction is executed* on a particular processor core during a cycle transition according to function *instrAddr*.

$$\begin{aligned}&\forall \mathit{ins} \in \mathit{InstrMemAddr} : \forall C_i \in \mathit{Cores} :\\ &\quad \mathit{Executes}_{\mathit{ins},C_i} = \{(s_1, s_2) \in \mathit{Cycle} \mid \mathit{ins} = \mathit{instrAddr}(s_1, C_i)\}\end{aligned} \tag{6.2}$$

We say that an instruction *retires* on processor core C_i during a cycle transition if an instance of this instruction completes its execution during the cycle transition on core C_i. The exact definition of this event depends on the implementation details of the micro-architecture and, thus, is omitted.

$$\begin{aligned}&\forall \mathit{ins} \in \mathit{InstrMemAddr} : \forall C_i \in \mathit{Cores} :\\ &\quad \mathit{Retires}_{\mathit{ins},C_i} \subseteq \mathit{Cycle}\end{aligned} \tag{6.3}$$

Note that, even with in-order retirement, multiple instructions may retire during the same cycle transition. Thus, an instruction may retire during a cycle transition although it was not officially considered as executed (i.e. it does not necessarily have to hold that $\mathit{Retires}_{\mathit{ins},C_i} \subseteq \mathit{Executes}_{\mathit{ins},C_i}$).

According to our view, a *program* is an identifier that is assigned to certain sub-sequences of the concrete traces occurring when considering the global operation of the hardware platform as a system. There is a set *Programs* of all program identifiers of the considered system.

$$\mathit{prog} \in \mathit{Programs} \tag{6.4}$$

In our simple program model, each program is assigned to the subset of the memory addresses that correspond to its program instructions. This assignment is performed by the function *allInstr* in such a way that the address space is partitioned into the sets of instructions of the different programs.

$$\mathit{allInstr} : \mathit{Programs} \rightarrow \mathcal{P}(\mathit{InstrMemAddr}) \tag{6.5}$$

$$\dot{\bigcup_{\mathit{prog} \in \mathit{Programs}}} \mathit{allInstr}(\mathit{prog}) = \mathit{InstrMemAddr} \tag{6.6}$$

A particular *program is executed* on a particular processor core during a cycle transition if and only if one of its instructions is executed on the core in question during the cycle transition.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad Executes_{prog,C_i} = \bigcup_{ins \in allInstr(prog)} Executes_{ins,C_i} \end{aligned} \tag{6.7}$$

Moreover, each program is assigned a set of start instructions and a set of end instructions from its program instructions.

$$startInstr : Programs \to \mathcal{P}(InstrMemAddr) \tag{6.8}$$

$$\forall prog \in Programs : startInstr(prog) \subseteq allInstr(prog) \tag{6.9}$$

$$endInstr : Programs \to \mathcal{P}(InstrMemAddr) \tag{6.10}$$

$$\forall prog \in Programs : endInstr(prog) \subseteq allInstr(prog) \tag{6.11}$$

A *program start* of a particular program is an event that occurs when the state of the system switches from a different instruction to a start instruction of the considered program during a cycle transition. A program start event also occurs when a program start instruction retires although it is not officially considered as executed. Note that a program start event is specific to a particular processor core.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad Start_{prog,C_i} = \{(s_1, s_2) \in Cycle \mid [instrAddr(s_2, C_i) \in startInstr(prog) \\ &\qquad\qquad \wedge\, instrAddr(s_2, C_i) \neq instrAddr(s_1, C_i)] \\ &\qquad\qquad \vee\, [\exists ins \in startInstr(prog) : \\ &\qquad\qquad\quad (s_1, s_2) \notin Executes_{ins,C_i} \wedge (s_1, s_2) \in Retires_{ins,C_i}]\} \end{aligned} \tag{6.12}$$

A *program end* event is defined similarly. In contrast to a program start event, however, it only occurs when an end instruction of the considered program retires.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad End_{prog,C_i} = \{(s_1, s_2) \in Retires_{ins,C_i} \mid ins \in endInstr(prog)\} \end{aligned} \tag{6.13}$$

Let *Traces* be the set of global concrete traces of the hardware system as defined in equation (5.4). It is defined based on the set S_{init} of initial system states and the cycle transition relation *Cycle*. From the set of global concrete traces, we can extract the set of initial states of program execution runs of a particular program. A target state of a cycle transition during which a corresponding program start event occurs is an initial state of a program execution run. Moreover, an initial state of the global system is also an initial state of a program execution run if it is assigned to the execution of a start instruction of the corresponding program.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad InitStates_{prog,C_i} = \{t(x+1) \mid t \in Traces \wedge x \in \mathbb{N}_{<len(t)} \\ &\qquad\qquad \wedge\, Start_{prog,C_i}(t, x)\} \\ &\qquad\qquad \cup\, \{s \in S_{init} \mid instrAddr(s, C_i) \in startInstr(prog)\} \end{aligned} \tag{6.14}$$

Based on these initial program states, we define the set of *program execution runs* of a particular program when executed on a particular processor core. Analogous to *Traces*, we define it as the set of all possible prefixes. An execution run prefix is a finite sequence of system states that begins in an initial program state. A program end event may only occur during the last cycle transition of an execution run prefix.

$$\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\quad \mathit{ExecRuns}_{\mathit{prog},C_i} = \{t \in \mathit{Sequences} \mid t(0) \in \mathit{InitStates}_{\mathit{prog},C_i} \\
&\qquad\qquad\qquad\qquad \wedge [\forall x \in \mathbb{N}_{<len(t)-1} : \neg \mathit{End}_{\mathit{prog},C_i}(t,x)]\}
\end{aligned} \tag{6.15}$$

Note that these execution run prefixes are by definition a subset of the general set *Sequences* (cf. equation (5.3)) of state sequences possible according to relation *Cycle*. Thus, the length of an execution run prefix as well as the shorthand to argue about the events at a particular position of an execution run prefix are defined by equations (5.5), (5.9), and (5.10). Also note that we already use the length and this shorthand in equation (6.15).

The program execution run prefixes of a program *prog* executed on processor core C_i can be seen as a system on its own. Thus, we also might know a set $\mathit{Prop}_{\mathit{prog},C_i}$ of system properties for this system. In this context, we can also refer to it as the set of program properties.

$$\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\quad \forall t \in \mathit{ExecRuns}_{\mathit{prog},C_i} : \forall P_k \in \mathit{Prop}_{\mathit{prog},C_i} : P_k(t)
\end{aligned} \tag{6.16}$$

Instruction set architectures (ISAs) typically provide instructions or sequences of instructions that drain the pipeline of a processor core (e.g. `sync` or `isync` on PowerPC [Diefendorff et al., 1994; Diefendorff and Silha, 1994]). In order to be able to analyze programs independently of each other, we rely on the *software convention* that the pipeline is drained at the end of each program execution run and whenever the control is passed on from one program to another. As a consequence, two instructions of different programs never retire at the same time on the same core.

$$\begin{aligned}
&\forall C_i \in \mathit{Cores} : \forall \mathit{prog}_1 \in \mathit{Programs} : \forall \mathit{prog}_2 \in \mathit{Programs} \setminus \{\mathit{prog}_1\} : \\
&\quad (\bigcup_{\mathit{ins} \in \mathit{allInstr}(\mathit{prog}_1)} \mathit{Retires}_{\mathit{ins},C_i}) \cap (\bigcup_{\mathit{ins} \in \mathit{allInstr}(\mathit{prog}_2)} \mathit{Retires}_{\mathit{ins},C_i}) = \emptyset
\end{aligned} \tag{6.17}$$

As a further consequence of this software convention, a start instruction and an end instruction of the same program never retire at the same time on the same core.

$$\begin{aligned}
&\forall C_i \in \mathit{Cores} : \forall \mathit{prog} \in \mathit{Programs} : \\
&\quad (\bigcup_{\mathit{ins} \in \mathit{startInstr}(\mathit{prog})} \mathit{Retires}_{\mathit{ins},C_i}) \cap (\bigcup_{\mathit{ins} \in \mathit{endInstr}(\mathit{prog})} \mathit{Retires}_{\mathit{ins},C_i}) = \emptyset
\end{aligned} \tag{6.18}$$

Note that the *program execution model* presented in this section naturally supports that some of the programs serve as *task schedulers*. Thus, we can realize dynamic task scheduling in this program execution model. The only restriction is that it does not support multiple active instances of the same program on the same processor core at the same time (as it does not keep track of which cycle transition belongs to which instance of a program—this is a design choice to keep things simple, a corresponding extension is possible but beyond the scope of this thesis). As a simple workaround, however, we can support up to x active instances of a particular program on the same processor core at the same time by creating x dedicated copies of the program in memory. Thus, we are confident that our program execution model is general enough for a wide range of real-time applications.

6.2. Exact Event Bounds

We define a shorthand event to argue about the situation that a particular event happens during the execution of a particular program on a particular processor core.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall t \in \mathit{Sequences} : \forall E \in \mathit{Events} :\\ &\quad \mathit{Event}_{\mathit{prog},C_i,E} = \mathit{Executes}_{\mathit{prog},C_i} \cap E\end{aligned} \tag{6.19}$$

The number of occurrences of a particular event E during the execution of program *prog* on core C_i within a sequence of system states is defined as follows.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall t \in \mathit{Sequences} : \forall E \in \mathit{Events} :\\ &\quad \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) = \left|\{x \in \mathbb{N}_{<\mathit{len}(t)} \mid \mathit{Event}_{\mathit{prog},C_i,E}(t,x)\}\right|\end{aligned} \tag{6.20}$$

We accordingly specify the maximal number of occurrences of a particular event E during any execution run of a particular program *prog* on core C_i.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} :\\ &\quad \mathit{Maximum}_{\mathit{prog},C_i,E} = \max_{t \in \mathit{ExecRuns}_{\mathit{prog},C_i}} \mathit{numEvOccur}(\mathit{prog}, C_i, t, E)\end{aligned} \tag{6.21}$$

The value of $\mathit{Maximum}_{\mathit{prog},C_i,E}$ is undefined if $\mathit{ExecRuns}_{\mathit{prog},C_i} = \emptyset$ (i.e. program *prog* is never executed on core C_i). Moreover, it is undefined if the maximal number of event occurrences over all execution runs cannot be bounded from above.

The WCET of a particular program when executed on a particular processor core can be seen as such a maximum. Essentially, for the WCET, we consider each cycle transition during which the considered program is executed. Thus, we choose to maximize the event *Cycle*.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} :\\ &\quad \mathit{WCET}_{\mathit{prog},C_i} = \mathit{Maximum}_{\mathit{prog},C_i,\mathit{Cycle}}\end{aligned} \tag{6.22}$$

In principle, we could specify $\mathit{Minimum}_{\mathit{prog},C_i,E}$ analogously to $\mathit{Maximum}_{\mathit{prog},C_i,E}$. However, recall that the set $\mathit{ExecRuns}_{\mathit{prog},C_i}$ contains all finite prefixes of the possible execution runs of program *prog* on core C_i. In particular, if the set is not empty, it also contains prefixes of length zero. Thus, such a specification of $\mathit{Minimum}_{\mathit{prog},C_i,E}$ would have a value of zero whenever it has a defined value. A lower bound value of zero, however, is maximally pessimistic and, thus, essentially useless in most cases. Therefore, we need a specification of $\mathit{Minimum}_{\mathit{prog},C_i,E}$ that is not based on all members of $\mathit{ExecRuns}_{\mathit{prog},C_i}$.

To this end, we further distinguish the members of $\mathit{ExecRuns}_{\mathit{prog},C_i}$. Some of its members are *terminated program execution runs*. This means that they end on a program end event.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} :\\ &\quad \mathit{ExecRuns}^{\mathit{term}}_{\mathit{prog},C_i} = \{t \in \mathit{ExecRuns}_{\mathit{prog},C_i} \mid \mathit{len}(t) \geq 1 \wedge \mathit{End}_{\mathit{prog},C_i}(t, \mathit{len}(t) - 1)\}\end{aligned} \tag{6.23}$$

For the remainder of this thesis, we assume that the concrete system has at least one successor state for every possible system state. Intuitively, this means that the system must not get stuck.

$$\forall s \in S : \exists s' \in S : (s, s') \in \mathit{Cycle} \tag{6.24}$$

Note that this does not restrict the generality of our model. We might still have a system state that logically correspond to a system termination if its set of successors according to *Cycle* only contains itself.

The relation *PrefixOf* captures that a sequences of system states is a *prefix* of another sequence of system states.

$$\begin{aligned}\mathit{PrefixOf} = \{(t_1, t_2) \in \mathit{Sequences} \times \mathit{Sequences} \mid \mathit{len}(t_1) \leq \mathit{len}(t_2) \\ \wedge \forall x \in \mathbb{N}_{\leq \mathit{len}(t_1)} : t_1(x) = t_2(x)\}\end{aligned} \tag{6.25}$$

A prefix of a particular sequence of system states is also a *strict prefix* of the sequence if and only if it is shorter than the sequence.

$$\mathit{StrictPrefixOf} = \{(t_1, t_2) \in \mathit{PrefixOf} \mid \mathit{len}(t_1) < \mathit{len}(t_2)\} \tag{6.26}$$

Intuitively, the strict prefixes of the terminated execution runs do not have to be considered in the specification of $\mathit{Minimum}_{prog,C_i,E}$ as they would only introduce unnecessary pessimism. However, in general, $\mathit{ExecRuns}_{prog,C_i}$ can contain more than only terminated execution runs and their prefixes. Programs that do not terminate in all situation may allow for execution run prefixes from which no program end event can be reached. We refer to them as *diverging execution run prefixes*. As every system state is guaranteed to have a successor (cf. equation (6.24)), we can safely calculate the diverging execution run prefixes by removing the terminated execution runs and all of their prefixes from $\mathit{ExecRuns}_{prog,C_i}$.

$$\begin{aligned}&\forall prog \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\ &\quad \mathit{ExecRuns}^{diverg}_{prog,C_i} = \mathit{ExecRuns}_{prog,C_i} \setminus \bigcup_{t \in \mathit{ExecRuns}^{term}_{prog,C_i}} \{t' \mid (t', t) \in \mathit{PrefixOf}\}\end{aligned} \tag{6.27}$$

For the specification of $\mathit{Minimum}_{prog,C_i,E}$, we only need to take into account those diverging execution run prefixes that end on event E during the execution of program *prog* on core C_i and allow to be continued arbitrarily long without a further occurrence of event E during the execution of program *prog* on core C_i.

$$\begin{aligned}&\forall prog \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} : \\ &\quad \mathit{ExecRuns}^{diverg,end}_{prog,C_i,E} = \{t \in \mathit{ExecRuns}^{diverg}_{prog,C_i} \mid \\ &\qquad [\mathit{len}(t) = 0 \vee (\mathit{len}(t) > 0 \wedge \mathit{Event}_{prog,C_i,E}(t, \mathit{len}(t) - 1))] \wedge \\ &\qquad \forall n \in \mathbb{N} : \exists t' \in \mathit{ExecRuns}^{diverg}_{prog,C_i} : \\ &\qquad\quad (t, t') \in \mathit{PrefixOf} \wedge \\ &\qquad\quad \mathit{len}(t') = \mathit{len}(t) + n \wedge \\ &\qquad\quad \forall x \in \mathbb{N}_{\geq \mathit{len}(t)} \cap \mathbb{N}_{< \mathit{len}(t')} : \\ &\qquad\qquad \neg \mathit{Event}_{prog,C_i,E}(t', x)\}\end{aligned} \tag{6.28}$$

Finally, we specify $\mathit{Minimum}_{prog,C_i,E}$ based on this subset of diverging execution run prefixes and the set of terminated execution runs. We refer to this combined subset of the execution run prefixes as *minimum-relevant* execution run prefixes.

$$\begin{aligned}&\forall prog \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} : \\ &\quad \mathit{ExecRuns}^{min\text{-}relev}_{prog,C_i,E} = \mathit{ExecRuns}^{term}_{prog,C_i} \cup \mathit{ExecRuns}^{diverg,end}_{prog,C_i,E}\end{aligned} \tag{6.29}$$

$$\begin{aligned}&\forall prog \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} : \\ &\quad \mathit{Minimum}_{prog,C_i,E} = \min_{t \in \mathit{ExecRuns}^{min\text{-}relev}_{prog,C_i,E}} \mathit{numEvOccur}(prog, C_i, t, E)\end{aligned} \tag{6.30}$$

The value of $\mathit{Minimum}_{prog,C_i,E}$ is undefined if $\mathit{ExecRuns}_{prog,C_i} = \emptyset$ (i.e. program *prog* is never executed on core C_i). Moreover, it is undefined if the minimal number of event occurrences over all execution runs cannot be exactly bounded from below (i.e. the program is guaranteed to diverge and to produce an unbounded amount of event occurrences).

The best-case execution time (BCET) of a particular program when executed on a particular processor core can be seen as such a minimum. We obtain it by choosing *Cycle* as the minimized event.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad BCET_{prog,C_i} = Minimum_{prog,C_i,Cycle}\end{aligned} \tag{6.31}$$

Note that—due to the definition of $ExecRuns_{prog,C_i}$ (cf. equations (6.15) and (5.3))—the BCET is special in the sense that $ExecRuns^{diverg,end}_{prog,C_i,Cycle}$ is guaranteed to coincide with the empty set. Thus, we can simplify the definition of the BCET in a way that no diverging execution run prefixes are involved.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad BCET_{prog,C_i} = \min_{t \in ExecRuns^{term}_{prog,C_i}} numEvOccur(prog, C_i, t, Cycle)\end{aligned} \tag{6.32}$$

This implies that, for the calculation of a BCET bound as a special case, it is safe to only take into account the terminated execution runs—even for programs that can diverge.

6.3. Event Bounds Based on Sequences of Abstract States

Section 5.2 introduced the approximation by sequences of abstract states. Moreover, it provided formal evidence that this way of approximation soundly overapproximates the global concrete traces of the system.

In this section, we use sequences of abstract states in order to soundly approximate the exact event bounds for program runs presented in the previous section. Recall that $(\widehat{Traces}, \gamma_{trace})$ is an abstract model of the set *Traces* of global concrete traces of the system. Now, we analogously design $(\widehat{ExecRuns}_{prog,C_i}, \gamma_{trace,prog,C_i})$ as an abstract model of $ExecRuns_{prog,C_i}$.

To this end, first, we choose a set of initial abstract states of a program execution run per combination of program and processor core. This set has to describe the initial system states of any actual program execution run.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad \widehat{InitStates}_{prog,C_i} \subseteq \widehat{S}\end{aligned} \tag{6.33}$$

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad \bigcup_{\widehat{s_i} \in \widehat{InitStates}_{prog,C_i}} \gamma(\widehat{s_i}) \supseteq InitStates_{prog,C_i}\end{aligned} \tag{6.34}$$

Next, we choose a set $\widehat{Traces}_{prog,C_i}$ per program *prog* and core C_i.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad \widehat{Traces}_{prog,C_i} \subseteq \widehat{Sequences}\end{aligned} \tag{6.35}$$

These sets have to fulfill the following two criteria. Note that these criteria are analogous to criteria (5.C1) and (5.C2). We just replaced $\widehat{Traces}$ by $\widehat{Traces}_{prog,C_i}$ and $\widehat{S_{init}}$ by $\widehat{InitStates}_{prog,C_i}$. Moreover, we added the universal quantifiers for all programs and all cores.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad \forall \widehat{s_i} \in \widehat{InitStates}_{prog,C_i} : \exists (\widehat{t}, \widehat{u}) \in \widehat{Traces}_{prog,C_i} : len((\widehat{t}, \widehat{u})) = 0 \land \widehat{t}(0) = \widehat{s_i}\end{aligned} \tag{6.C1}$$

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores : \\
&\quad \forall(\widehat{t_1}, \widehat{u_1}) \in \widehat{Traces_{prog,C_i}} : \\
&\quad\quad \forall(\widehat{u_1}(len((\widehat{t_1}, \widehat{u_1}))), \widehat{s_c}) \in \widehat{Cycle} : \\
&\quad\quad\quad \exists(\widehat{t_2}, \widehat{u_2}) \in \widehat{Traces_{prog,C_i}} : \\
&\quad\quad\quad\quad len((\widehat{t_2}, \widehat{u_2})) = 1 + len((\widehat{t_1}, \widehat{u_1})) \ \wedge \\
&\quad\quad\quad\quad (\forall x \in \mathbb{N}_{\leq len((\widehat{t_1}, \widehat{u_1}))} : \widehat{t_2}(x) = \widehat{t_1}(x) \wedge \widehat{u_2}(x) = \widehat{u_1}(x)) \ \wedge \\
&\quad\quad\quad\quad \widehat{t_2}(len((\widehat{t_2}, \widehat{u_2}))) = \widehat{s_c}
\end{aligned} \tag{6.C2}$$

Based on $\widehat{Traces_{prog,C_i}}$, we define the set $\widehat{ExecRuns_{prog,C_i}}$ of abstract program execution run prefixes of a particular program when executed on a particular processor core. However, we only consider those members of $\widehat{Traces_{prog,C_i}}$ for which a guaranteed program end event (must-event, cf. equation (5.39)) does not occur before the last abstract cycle transition.

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores : \\
&\quad \widehat{ExecRuns_{prog,C_i}} = \{\widehat{t} \in \widehat{Traces_{prog,C_i}} \mid \forall x \in \mathbb{N}_{<len(\widehat{t})-1} : \neg \widehat{End^{LB}_{prog,C_i}}(\widehat{t}, x)\}
\end{aligned} \tag{6.36}$$

Note that these prefixes are by definition a subset of the general set $\widehat{Sequences}$ (cf. equation (5.18)) of abstract state sequences possible according to relation $\widehat{Cycle}$. Thus, the length of a prefix as well as the shorthand to argue about the abstract events at a particular position of a prefix are defined by equations (5.19), (5.36), and (5.37). Also note that we already use the length and this shorthand in equation (6.36).

Next, we define the set $Spurious_{prog,C_i}$, which contains the spurious constructs that are not contained in $ExecRuns_{prog,C_i}$. Its definition is analog to the definition of $Spurious$ (equation (5.24)) in Section 5.2. The only difference is that, this time, we replace S_{init} by $InitStates_{prog,C_i}$.

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores : \\
&\quad Spurious_{prog,C_i} = \{(t, u) \in (\mathbb{N}_{<n} \to S) \times ((\mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n}) \to S) \\
&\quad\quad \mid n \in \mathbb{N} \wedge [\forall x \in \mathbb{N}_{<n} : (t(x), u(x+1)) \in Cycle] \\
&\quad\quad \wedge [t(0) \notin InitStates_{prog,C_i} \\
&\quad\quad\quad \vee \exists x \in (\mathbb{N}_{\geq 1} \cap \mathbb{N}_{<n}) : t(x) \neq u(x)]\}
\end{aligned} \tag{6.37}$$

Since $Spurious_{prog,C_i}$ is defined analogously to $Spurious$, the length of its members and the shorthand to argue about the events at a particular position its members are also defined analogously (cf. equations (5.25), (5.26), and (5.27)) and, thus, not explicitly defined here.

The number of occurrences of a particular event E during the execution of program $prog$ on core C_i within a member of $Spurious_{prog,C_i}$ is defined as follows.

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores : \forall t \in Spurious_{prog,C_i} : \forall E \in Events : \\
&\quad numEvOccur(prog, C_i, t, E) = \left|\{x \in \mathbb{N}_{<len(t)} \mid Event_{prog,C_i,E}(t, x)\}\right|
\end{aligned} \tag{6.38}$$

Finally, we define the mapping from members of $\widehat{Traces_{prog,C_i}}$ to actual and spurious execution run prefixes. Its definition is analog to the definition of γ_{trace} (cf. equation (5.31)).

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores : \\
&\quad \gamma_{trace,prog,C_i} \in (\widehat{Traces_{prog,C_i}} \to \mathcal{P}(Sequences \cup Spurious_{prog,C_i}))
\end{aligned} \tag{6.39}$$

$$
\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\quad \gamma_{\mathit{trace},\mathit{prog},C_i}((\widehat{t},\widehat{u})) = \{t \in \mathit{Sequences} \mid t(0) \in \mathit{InitStates}_{\mathit{prog},C_i} \\
&\qquad\qquad \wedge\ \mathit{len}((\widehat{t},\widehat{u})) = \mathit{len}(t) \\
&\qquad\qquad \wedge\ \forall x \in \mathbb{N}_{\leq \mathit{len}(t)} : t(x) \in \gamma(\widehat{t}(x))\} \cup \\
&\qquad \{(t,u) \in \mathit{Spurious}_{\mathit{prog},C_i} \mid \mathit{len}((\widehat{t},\widehat{u})) = \mathit{len}((t,u)) \\
&\qquad\qquad \wedge\ [\forall x \in \mathbb{N}_{< \mathit{len}(t)} : t(x) \in \gamma(\widehat{u}(x))] \\
&\qquad\qquad \wedge\ \forall x \in (\mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq \mathit{len}(t)}) : u(x) \in \gamma(\widehat{t}(x))\}
\end{aligned}
\tag{6.40}
$$

It follows that $(\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i}, \gamma_{\mathit{trace},\mathit{prog},C_i})$ is an abstract model of $\mathit{ExecRuns}_{\mathit{prog},C_i}$ as it provides an overapproximation of this set. For a detailed proof of this statement, we refer to page 277.

$$
\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\bigcup_{\widehat{t} \in \widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i}} \gamma_{\mathit{trace},\mathit{prog},C_i}(\widehat{t}) \supseteq \mathit{ExecRuns}_{\mathit{prog},C_i}
\end{aligned}
\tag{6.41}
$$

The program properties from set $\mathit{Prop}_{\mathit{prog},C_i}$ can be safely lifted to the sequences of abstract states contained in $\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i}$ by applying the lifting rules presented in Section 5.2. Let $\widehat{P_k}$ be the lifted version of program property P_k that is obtained by applying the aforementioned lifting rules. We can safely use the lifted versions of the program properties in order to detect infeasible members of $\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i}$.

$$
\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\quad \widehat{\mathit{LessExecRuns}}_{\mathit{prog},C_i} = \{\widehat{t} \in \widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i} \mid \forall P_k \in \mathit{Prop}_{\mathit{prog},C_i} : \widehat{P_k}(\widehat{t})\}
\end{aligned}
\tag{6.42}
$$

As a consequence, $(\widehat{\mathit{LessExecRuns}}_{\mathit{prog},C_i}, \gamma_{\mathit{trace},\mathit{prog},C_i})$ is also an abstract model of the actual execution run prefixes. The proof of this statement is a slight extension of the proof of statement (6.41). Thus, we only present a brief proof sketch on page 279.

$$
\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\bigcup_{\widehat{t} \in \widehat{\mathit{LessExecRuns}}_{\mathit{prog},C_i}} \gamma_{\mathit{trace},\mathit{prog},C_i}(\widehat{t}) \supseteq \mathit{ExecRuns}_{\mathit{prog},C_i}
\end{aligned}
\tag{6.43}
$$

Analogously to *numEvOccur*, we can bound the number of occurrences of a particular event E during the execution of program *prog* on core C_i within any member of $\widehat{\mathit{Sequences}}$ from above (UB) and from below (LB).

$$
\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall \widehat{t} \in \widehat{\mathit{Sequences}} : \forall E \in \mathit{Events} : \\
&\quad \forall \mathit{BD} \in \{\mathit{UB}, \mathit{LB}\} : \\
&\quad \widehat{\mathit{numEvOccur}}(\mathit{prog}, C_i, \widehat{t}, E, \mathit{BD}) = \left|\{x \in \mathbb{N}_{< \mathit{len}(\widehat{t})} \mid \widehat{\mathit{Event}}^{\mathit{BD}}_{\mathit{prog},C_i,E}(\widehat{t}, x)\}\right|
\end{aligned}
\tag{6.44}
$$

We use $\widehat{\mathit{numEvOccur}}$ to specify an upper bound $\widehat{\mathit{Maximum}}_{\mathit{prog},C_i,E}$ on the number of event occurrences based on $\widehat{\mathit{LessExecRuns}}_{\mathit{prog},C_i}$.

$$
\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} : \\
&\quad \widehat{\mathit{Maximum}}_{\mathit{prog},C_i,E} = \max_{\widehat{t} \in \widehat{\mathit{LessExecRuns}}_{\mathit{prog},C_i}} \widehat{\mathit{numEvOccur}}(\mathit{prog}, C_i, \widehat{t}, E, \mathit{UB})
\end{aligned}
\tag{6.45}
$$

In case $\widehat{Maximum}_{prog,C_i,E}$ has a defined value (i.e. $\in \mathbb{N}$), this value is guaranteed to be an upper bound on the possible numbers of event occurrences in any execution run of the actual system. For a formal proof of this statement, we refer to page 279.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \forall E \in Events : \\ &\quad \widehat{Maximum}_{prog,C_i,E} \in \mathbb{N} \Rightarrow \\ &\quad \forall t \in ExecRuns_{prog,C_i} : numEvOccur(prog, C_i, t, E) \leq \widehat{Maximum}_{prog,C_i,E} \end{aligned} \tag{6.46}$$

Next, we set up the machinery that is needed to specify a corresponding lower bound $\widehat{Minimum}_{prog,C_i,E}$ on the number of event occurrences. Similarly to the specification of the exact lower bound $Minimum_{prog,C_i,E}$, this requires to define particular subsets of $\widehat{ExecRuns}_{prog,C_i}$. First, we define a subset of $\widehat{ExecRuns}_{prog,C_i}$ that describes all terminated program execution runs.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad \widehat{ExecRuns}^{term}_{prog,C_i} = \{\widehat{t} \in \widehat{ExecRuns}_{prog,C_i} \mid len(\widehat{t}) \geq 1 \wedge \widehat{End}^{UB}_{prog,C_i}(\widehat{t}, len(\widehat{t}) - 1)\} \end{aligned} \tag{6.47}$$

The (optionally strict) prefix relation between sequences of abstract states is defined analogously to the corresponding relation between sequences of system states.

$$\begin{aligned} \widehat{PrefixOf} = \{((\widehat{t_1}, \widehat{u_1}), (\widehat{t_2}, \widehat{u_2})) \in \widehat{Sequences} \times \widehat{Sequences} \mid len((\widehat{t_1}, \widehat{u_1})) \leq len((\widehat{t_2}, \widehat{u_2})) \\ \wedge \forall x \in \mathbb{N}_{\leq len((\widehat{t_1}, \widehat{u_1}))} : \widehat{t_1}(x) = \widehat{t_2}(x) \wedge \widehat{u_1}(x) = \widehat{u_2}(x)\} \end{aligned} \tag{6.48}$$

$$\widehat{StrictPrefixOf} = \{(\widehat{t_1}, \widehat{t_2}) \in \widehat{PrefixOf} \mid len(\widehat{t_1}) < len(\widehat{t_2})\} \tag{6.49}$$

In the following, we define a set of sequences of abstract states that overapproximates $ExecRuns^{diverg,end}_{prog,C_i,E}$.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \forall E \in Events : \\ &\quad \widehat{ExecRuns}^{diverg,end}_{prog,C_i,E} = \{\widehat{t} \in \widehat{ExecRuns}_{prog,C_i} \mid \\ &\qquad [len(\widehat{t}) = 0 \vee (len(\widehat{t}) > 0 \wedge \widehat{Event}^{UB}_{prog,C_i,E}(\widehat{t}, len(\widehat{t}) - 1))] \wedge \\ &\qquad \forall n \in \mathbb{N} : \exists \widehat{t'} \in \widehat{ExecRuns}_{prog,C_i} : \\ &\qquad\quad (\widehat{t}, \widehat{t'}) \in \widehat{PrefixOf} \wedge \\ &\qquad\quad len(\widehat{t'}) = len(\widehat{t}) + n \wedge \\ &\qquad\quad \forall x \in \mathbb{N}_{\geq len(\widehat{t})} \cap \mathbb{N}_{< len(\widehat{t'})} : \\ &\qquad\qquad \neg \widehat{Event}^{LB}_{prog,C_i,E}(\widehat{t'}, x)\} \end{aligned} \tag{6.50}$$

Based on these ingredients, we define the following set of sequences of abstract states.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \forall E \in Events : \\ &\quad \widehat{ExecRuns}^{min\text{-}relev}_{prog,C_i,E} = \widehat{ExecRuns}^{term}_{prog,C_i} \cup \widehat{ExecRuns}^{diverg,end}_{prog,C_i,E} \end{aligned} \tag{6.51}$$

It overapproximates the minimum-relevant execution run prefixes. For a formal proof of this statement, we refer to page 280.

$$\begin{aligned} &\forall prog \in Programs : \forall C_i \in Cores : \\ &\quad \bigcup_{\widehat{t} \in \widehat{ExecRuns}^{min\text{-}relev}_{prog,C_i,E}} \gamma_{trace,prog,C_i}(\widehat{t}) \supseteq ExecRuns^{min\text{-}relev}_{prog,C_i,E} \end{aligned} \tag{6.52}$$

We can safely use lifted versions of the program properties from set $\mathit{Prop}_{\mathit{prog},C_i}$ in order to detect infeasible members of $\widehat{\mathit{ExecRuns}}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E}$.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} :\\ &\quad \widehat{\mathit{LessExecRuns}}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E} = \{\widehat{t} \in \widehat{\mathit{ExecRuns}}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E} \mid \forall P_k \in \mathit{Prop}_{\mathit{prog},C_i} : \widehat{P_k}(\widehat{t})\}\end{aligned} \tag{6.53}$$

Thus, $\widehat{\mathit{LessExecRuns}}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E}$ also overapproximates the minimum-relevant execution run prefixes. The formal proof of this statement can be performed by a slight extension to the proof of statement (6.52). Thus, we only sketch it on page 284.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} :\\ &\quad \bigcup_{\widehat{t} \in \widehat{\mathit{LessExecRuns}}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E}} \gamma_{\mathit{trace},\mathit{prog},C_i}(\widehat{t}) \supseteq \mathit{ExecRuns}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E}\end{aligned} \tag{6.54}$$

Finally, we define $\widehat{\mathit{Minimum}}_{\mathit{prog},C_i,E}$ based on this set of sequences of abstract states.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} :\\ &\quad \widehat{\mathit{Minimum}}_{\mathit{prog},C_i,E} = \min_{\widehat{t} \in \widehat{\mathit{LessExecRuns}}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E}} \widehat{\mathit{numEvOccur}}(\mathit{prog}, C_i, \widehat{t}, E, \mathit{LB})\end{aligned} \tag{6.55}$$

In case $\widehat{\mathit{Minimum}}_{\mathit{prog},C_i,E}$ has a defined value (i.e. $\in \mathbb{N}$), this value is guaranteed to be a lower bound on the possible numbers of event occurrences in any execution run of the actual system. The formal proof of this statement is very similar to the proof of statement (6.46) on page 279 and, thus, omitted.

$$\begin{aligned}&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \forall E \in \mathit{Events} :\\ &\quad \widehat{\mathit{Minimum}}_{\mathit{prog},C_i,E} \in \mathbb{N} \Rightarrow\\ &\quad \forall t \in \mathit{ExecRuns}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E} : \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) \geq \widehat{\mathit{Minimum}}_{\mathit{prog},C_i,E}\end{aligned} \tag{6.56}$$

6.4. Event Bounds Based on Paths through a Graph

In this section, we present a detailed graph representation of the results of a micro-architectural analysis. Subsequently, we define a subsumption relation for graphs. It is defined in a way that any graph subsuming the detailed graph can soundly be used for the calculation of event bounds. Based on this, we specify a generic family of graphs that subsume the detailed graph. Last but not least, we present soundness criteria for graph transformations.

Before we start with the actual content of this section, we introduce some notational conventions. In Section 5.3, we only argue about a single graph. The current chapter, however, argues about different graphs. In order to tell them apart, the constituents of each graph G^A are annotated with a corresponding superscript.

$$G^A = (\mathit{Nodes}^A, \mathit{Nodes}^A_{\mathit{start}}, \mathit{Nodes}^A_{\mathit{end}}, \mathit{Edges}^A) \tag{6.57}$$

The relations between these constituents per graph shall be as defined in Section 5.3. The corresponding sets $\widehat{SubPaths}^A$, $\widehat{RelPaths}^A$, and $\widehat{Paths}^A$ shall also be defined correspondingly. Moreover, we need a prefix relation on the subpaths per graph G^A. It shall be defined similar to the prefix relations on sequences of abstract states and system states.

$$\begin{aligned}\widehat{PrefixOf}^A_{path} = \{(\widehat{p_1}, \widehat{p_2}) \in \widehat{SubPaths}^A \times \widehat{SubPaths}^A \mid len(\widehat{p_1}) \leq len(\widehat{p_2}) \\ \wedge \forall x \in \mathbb{N}_{\leq len(\widehat{p_1})} : \widehat{p_1}(x) = \widehat{p_2}(x)\}\end{aligned} \tag{6.58}$$

$$\widehat{StrictPrefixOf}^A_{path} = \{(\widehat{p_1}, \widehat{p_2}) \in \widehat{PrefixOf}^A_{path} \mid len(\widehat{p_1}) < len(\widehat{p_2})\} \tag{6.59}$$

In contrast to Section 5.3, now, we have different sets $\widehat{Traces}_{prog,C_i}$ of sequences of abstract states depending on the program *prog* and the core C_i. Thus, we have different functions to map graph paths to sequences of abstract states per graph G^A, program *prog*, and core C_i. The definitions of these functions shall be analogous to the definition presented in Section 5.3.

$$\gamma^A_{path,prog,C_i} : \widehat{RelPaths}^A \rightarrow \mathcal{P}(\widehat{Traces}_{prog,C_i} \cup \widehat{SpuriousTraces}_{prog,C_i}) \tag{6.60}$$

Finally, note that—starting from this section—we omit the explicit universal quantifiers for programs *prog* and processor cores C_i for the sake of a better readability. Thus, you can assume all the following statements to implicitly hold for all programs and cores.

6.4.1. Control Flow and Control Flow Graphs

The *control flow* of an assembly-level program typically refers to the order in which the assembly instructions of the program are executed during a program run. In the micro-architecture of modern processors, however, there is typically more than one instruction processed at a time by a processor core. In this subsection, we formally define the connection between a concrete trace of a program's execution and the corresponding control flow.

We define a control flow as a sequence of instruction memory addresses. It is modeled as a finite, potentially empty list of instruction memory addresses.

$$InstrMemAddrList = \bigcup_{n \in \mathbb{N}} (\mathbb{N}_{<n} \rightarrow InstrMemAddr) \tag{6.61}$$

The length of such a list is naturally given by its number of entries.

$$\forall n \in \mathbb{N} : \forall list \in InstrMemAddrList \cap (\mathbb{N}_{<n} \rightarrow InstrMemAddr) : len(list) = n \tag{6.62}$$

Moreover, we define a few helper functions on the lists. Every non-empty list has an element at its front position and an element at its back position.

$$\forall list \in \bigcup_{n \in \mathbb{N}_{>0}} (\mathbb{N}_{<n} \rightarrow InstrMemAddr) : front(list) = list(0) \tag{6.63}$$

$$\forall list \in \bigcup_{n \in \mathbb{N}_{>0}} (\mathbb{N}_{<n} \rightarrow InstrMemAddr) : back(list) = list(len(list) - 1) \tag{6.64}$$

We define a function *popFront* that creates a new list by removing the first element of an existing non-empty list.

$$\forall list \in \bigcup_{n \in \mathbb{N}_{>0}} (\mathbb{N}_{<n} \rightarrow InstrMemAddr) : len(popFront(list)) = len(list) - 1 \tag{6.65}$$

$$\begin{aligned}\forall list \in \bigcup_{n \in \mathbb{N}_{>0}} (\mathbb{N}_{<n} \rightarrow InstrMemAddr) : \forall x \in \mathbb{N}_{<len(list)-1} : \\ popFront(list)(x) = list(x + 1)\end{aligned} \tag{6.66}$$

In case we are only interested in whether a particular address is contained in a list, it is convenient to transform the list to a set.

$$\begin{aligned}&\forall \mathit{list} \in \mathit{InstrMemAddrList}:\\ &\quad \mathit{toSet}(\mathit{list}) = \{\mathit{ins} \in \mathit{InstrMemAddr} \mid \exists x \in \mathbb{N}_{<\mathit{len}(\mathit{list})} : \mathit{list}(x) = \mathit{ins}\}\end{aligned} \tag{6.67}$$

We are able to create a new list by applying a filter criterion to an existing list.

$$\mathit{filter}: \mathit{InstrMemAddrList} \times (\mathit{InstrMemAddr} \to \{0,1\}) \to \mathit{InstrMemAddrList} \tag{6.68}$$

$$\mathit{len}(\mathit{filter}(\mathit{list}, \mathit{crit})) = \left|\{x \in \mathbb{N}_{<\mathit{len}(\mathit{list})} \mid \mathit{crit}(\mathit{list}(x))\}\right| \tag{6.69}$$

$$\mathit{filter}(\mathit{list}, \mathit{crit})(x) = \mathit{list}(\min\{y \in \mathbb{N}_{<\mathit{len}(\mathit{list})} \mid \sum_{z \in \mathbb{N}_{\leq y}} \mathit{crit}(\mathit{list}(z)) = x + 1\}) \tag{6.70}$$

A list of these lists is defined analogously.

$$\mathit{InstrMemAddrListList} = \bigcup_{n \in \mathbb{N}} (\mathbb{N}_{<n} \to \mathit{InstrMemAddrList}) \tag{6.71}$$

$$\forall n \in \mathbb{N} : \forall \mathit{list} \in \mathit{InstrMemAddrListList} \cap (\mathbb{N}_{<n} \to \mathit{InstrMemAddrList}) : \mathit{len}(\mathit{list}) = n \tag{6.72}$$

We concatenate the members of a list of lists in the following way.

$$\mathit{concat}: \mathit{InstrMemAddrListList} \to \mathit{InstrMemAddrList} \tag{6.73}$$

$$\mathit{len}(\mathit{concat}(\mathit{list})) = \sum_{x \in \mathbb{N}_{<\mathit{len}(\mathit{list})}} \mathit{len}(\mathit{list}(x)) \tag{6.74}$$

$$\begin{aligned}&\mathit{concat}(\mathit{list})(x) = \mathit{list}(y)(z)\\ &\quad \mathit{with}\ y = \min\{y' \in \mathbb{N}_{<\mathit{len}(\mathit{list})} \mid \sum_{y'' \in \mathbb{N}_{\leq y'}} \mathit{len}(\mathit{list}(y'')) \geq x + 1\}\\ &\quad \mathit{and}\ z = x + 1 - \sum_{y'' \in \mathbb{N}_{<y}} \mathit{len}(\mathit{list}(y''))\end{aligned} \tag{6.75}$$

We extract the control flow of a particular processor core from a concrete trace by considering which instructions retire in which order on the core. This results in the actual control flow as instructions are typically retired in order (cf. Section 6.1) and misspeculated instruction executions are not retired at all. However, due to performance-enhancing pipeline features as out-of-order execution, it is possible that multiple instructions retire during the same cycle transition of the system. Intuitively, this means that the execution of one instruction can be completely overlapped by the execution of its predecessor instruction.

Our current event-based notion of retirement is not able to determine which instruction retires earlier in the sense of control flow when multiple instructions retire during the same cycle transition. Thus, we additionally provide a detailed order between multiple instructions that retire during the same cycle transition. Note that the actual hardware also tracks this order as it has to prohibit out-of-order retirement. The order is given by a function $\mathit{retires}_{C_i}$ that maps every cycle transition to a list of instruction memory addresses that retire in list order on core C_i during the transition.

$$\forall C_i \in \mathit{Cores} : \mathit{retires}_{C_i} \in (\mathit{Cycle} \to \mathit{InstrMemAddrList}) \tag{6.76}$$

It is compatible to the event view on retirement (i.e. it retires the same instruction memory addresses per cycle transition).

$$\begin{aligned}&\forall C_i \in \mathit{Cores} : \forall (s_1, s_2) \in \mathit{Cycle}:\\ &\quad \mathit{toSet}(\mathit{retires}_{C_i}((s_1, s_2))) = \{\mathit{ins} \in \mathit{InstrMemAddr} \mid (s_1, s_2) \in \mathit{Retires}_{\mathit{ins}, C_i}\}\end{aligned} \tag{6.77}$$

With this machinery in place, we can extract the control flow of a processor core from a sequence of system states. To this end, we map each cycle transition of the sequence to its list of instructions retired on the core. If the last cycle transition of the sequence does not retire any instructions, we assign it to the next instruction that will retire on the core. Finally, we concatenate these lists in the order of their appearance in the sequence of system states in order to obtain the control flow of the core corresponding to the sequence.

$$\mathit{toContrFlow}_{C_i} : \mathit{Sequences} \rightarrow \mathit{InstrMemAddrList} \tag{6.78}$$

$$\begin{aligned}
&\mathit{toContrFlow}_{C_i}(t) = \mathit{concat}(\\
&\quad \lambda x \in \mathbb{N}_{<\mathit{len}(t)}. \begin{cases} \mathit{retires}_{C_i}((t(x), t(x+1))) & , \mathit{if}\ x < \mathit{len}(t) - 1 \vee \\ & \quad \mathit{len}(\mathit{retires}_{C_i}((t(x), t(x+1)))) > 0 \\ \lambda y \in \{0\}.\mathit{instrAddr}(t(\mathit{len}(t) - 1)) & , \mathit{else} \end{cases} \\
&)
\end{aligned} \tag{6.79}$$

We can apply a filter to reduce the extracted control to only the instructions belonging to a particular program *prog*.

$$\mathit{toContrFlow}_{\mathit{prog},C_i} : \mathit{Sequences} \rightarrow \mathit{InstrMemAddrList} \tag{6.80}$$

$$\mathit{toContrFlow}_{\mathit{prog},C_i}(t) = \mathit{filter}(\mathit{toContrFlow}_{C_i}(t), \lambda \mathit{ins}.\mathit{ins} \in \mathit{allInstr}(\mathit{prog})) \tag{6.81}$$

The set of all possible control flows of a particular program when executed on a particular processor core is defined based on the execution run prefixes of the program on the core. This relies on the assumption that only a single instance of each program may be active at a time per processor core (cf. last paragraph of Section 6.1). Otherwise, this set might also contain control flows that are only possible by multiple overlapping instances of the same program on the same core.

$$\mathit{ContrFlows}_{\mathit{prog},C_i} = \{\mathit{toContrFlow}_{\mathit{prog},C_i}(t) \mid t \in \mathit{ExecRuns}_{\mathit{prog},C_i}\} \tag{6.82}$$

For the instructions of a program, there shall be a partitioning into *basic blocks* per core. Each basic block is a sequence of instruction memory addresses.

$$\mathit{BasicBlocks}_{\mathit{prog},C_i} \in \mathcal{P}(\mathit{InstrMemAddrList}) \tag{6.83}$$

$$\dot{\bigcup_{bb \in \mathit{BasicBlocks}_{\mathit{prog},C_i}}} \mathit{toSet}(bb) = \mathit{allInstr}(\mathit{prog}) \tag{6.84}$$

Moreover, no basic block shall be empty or containing duplicates.

$$\forall bb \in \mathit{BasicBlocks}_{\mathit{prog},C_i} : \mathit{len}(bb) > 0 \tag{6.85}$$

$$\forall bb \in \mathit{BasicBlocks}_{\mathit{prog},C_i} : \mathit{len}(bb) = |\mathit{toSet}(bb)| \tag{6.86}$$

Whenever an instruction of a basic block appears at a particular position within a control flow of program *prog* on core C_i, the neighboring positions in the flow shall also be occupied by the corresponding other instructions in the basic block.

$$\begin{aligned}
&\forall bb \in \mathit{BasicBlocks}_{\mathit{prog},C_i} : \\
&\quad \forall bbPos \in \mathbb{N}_{<\mathit{len}(bb)} : \\
&\quad\quad \forall \mathit{flow} \in \mathit{ContrFlows}_{\mathit{prog},C_i} : \\
&\quad\quad\quad \forall \mathit{flowPos} \in \{x \in \mathbb{N}_{<\mathit{len}(\mathit{flow})} \mid \mathit{flow}(x) = bb(bbPos)\} : \\
&\quad\quad\quad\quad \forall \mathit{flowPos}' \in \{x \in \mathbb{N}_{<\mathit{len}(\mathit{flow})} \mid \mathit{flowPos} - bbPos \leq x < \mathit{flowPos} - bbPos + \mathit{len}(bb)\} : \\
&\quad\quad\quad\quad\quad \mathit{flow}(\mathit{flowPos}') = bb(\mathit{flowPos}' - (\mathit{flowPos} - bbPos))
\end{aligned} \tag{6.87}$$

Finally, each start instruction of the program shall occupy the front position of a basic block and each end instruction shall occupy the back position of a basic block.

$$\forall \mathit{ins} \in \mathit{startInstr}(\mathit{prog}) : \exists \mathit{bb} \in \mathit{BasicBlocks}_{\mathit{prog},C_i} : \mathit{ins} = \mathit{front}(\mathit{bb}) \tag{6.88}$$

$$\forall \mathit{ins} \in \mathit{endInstr}(\mathit{prog}) : \exists \mathit{bb} \in \mathit{BasicBlocks}_{\mathit{prog},C_i} : \mathit{ins} = \mathit{back}(\mathit{bb}) \tag{6.89}$$

A control flow graph (CFG) $G^{\mathit{cfg},\mathit{prog},C_i}$ is a graph that uses such a set of basic blocks as nodes.

$$\mathit{Nodes}^{\mathit{cfg},\mathit{prog},C_i} = \mathit{BasicBlocks}_{\mathit{prog},C_i} \tag{6.90}$$

Moreover, the edges and start nodes of $G^{\mathit{cfg},\mathit{prog},C_i}$ are chosen in a way that its relaxed paths safely overapproximate every control flow that program *prog* can produce on core C_i.

$$\begin{aligned}&\forall \mathit{flow} \in \mathit{ContrFlows}_{\mathit{prog},C_i} : \exists \widehat{p} \in \widehat{\mathit{RelPaths}^{\mathit{cfg},\mathit{prog},C_i}} :\\ &\quad \mathit{len}(\mathit{flow}) \leq \mathit{len}(\mathit{concat}(\widehat{p})) \wedge\\ &\quad \forall x \in \mathbb{N}_{<\mathit{len}(\mathit{flow})} : \mathit{flow}(x) = \mathit{concat}(\widehat{p})(x)\end{aligned} \tag{6.91}$$

Control flow graphs are the typical program representation in most compilers[1]. The instructions in a basic block are usually mapped to consecutive addresses in program memory. Moreover, compilers typically make sure that at most the last instruction in a basic block can manipulate the program counter.

Static WCET analysis is typically performed on a control flow graph of the program under analysis [Thesing, 2004; Ballabriga et al., 2010; Puaut and Hardy, 2014]. In case the program under analysis is only available in binary form, a control flow graph has to be reconstructed from the binary file [Theiling, 2000; Kästner and Wilhelm, 2002]. The prototype implementation of our analysis tool, in contrast, is integrated into the back-end of a compiler infrastructure [Jacobs et al., 2015] and, thus, can reuse the control flow graph still available at the end of the compilation process.

Next, we present a detailed graph representation of the results of a micro-architectural analysis and map its nodes to the basic blocks of a control flow graph of the program under analysis.

6.4.2. A Detailed Graph Representation

Intuitively, a micro-architectural analysis is performed by propagating abstract states through the basic blocks of a control flow graph of the program under analysis until a fixed point is reached. It starts from a set of initial abstract states that safely overapproximates the possible system states at program start.

In this section, we represent the fixed point of such a micro-architectural analysis as a detailed graph $G^{\mathit{detail},\mathit{prog},C_i}$ at the granularity of cycle transitions. We mostly see this detailed graph as a conceptual construction which is later used in the soundness argument for bound calculations on more coarse graph representations. But in principle, such a detailed graph representation can be constructed in a canonical way from the results of a micro-architectural analysis.

Every node of the detailed graph shall be mapped to an abstract state by the following function.

$$\mathit{toState} : \mathit{Nodes}^{\mathit{detail},\mathit{prog},C_i} \to \widehat{S} \tag{6.92}$$

For this detailed graph representation, we require two additional assumptions about the transition system on abstract states with respect to the processor core C_i under analysis.

$$\forall \widehat{s} \in \widehat{S} : \forall s_1, s_2 \in \gamma(\widehat{s}) : \mathit{instrAddr}(s_1, C_i) = \mathit{instrAddr}(s_2, C_i) \tag{6.93}$$

$$\begin{aligned}&\forall (\widehat{s_1}, \widehat{s_2}) \in \widehat{\mathit{Cycle}} : \forall (s_1, s_2), (s_1', s_2') \in \mathit{Cycle} \cap (\gamma(\widehat{s_1}) \times \gamma(\widehat{s_1})) :\\ &\quad \mathit{retires}_{C_i}((s_1, s_2)) = \mathit{retires}_{C_i}((s_1', s_2'))\end{aligned} \tag{6.94}$$

[1] https://gcc.gnu.org, http://llvm.org

Intuitively, they mean that the abstract states exactly model the instructions in the pipeline of the considered processor core [Thesing, 2004]. As a consequence, we can directly reuse the functions *instrAddr* and $\mathit{retires}_{C_i}$ for abstract states respectively cycle transitions between them.

$$\forall \widehat{s} \in \widehat{S} : \forall s \in \gamma(\widehat{s}) : \mathit{instrAddr}(s, C_i) = \mathit{instrAddr}(\widehat{s}, C_i) \tag{6.95}$$

$$\begin{gathered}\forall (\widehat{s_1}, \widehat{s_2}) \in \widehat{\mathit{Cycle}} : \forall (s_1, s_2) \in \mathit{Cycle} \cap (\gamma(\widehat{s_1}) \times \gamma(\widehat{s_1})) : \\ \mathit{retires}_{C_i}((s_1, s_2)) = \mathit{retires}_{C_i}((\widehat{s_1}, \widehat{s_2}))\end{gathered} \tag{6.96}$$

We continue the specification of the detailed graph by splitting the nodes of the graph into two disjoint partitions.

$$\mathit{Nodes}^{\mathit{detail,prog},C_i} = T^{\mathit{detail,prog},C_i} \mathbin{\dot{\cup}} U^{\mathit{detail,prog},C_i} \tag{6.97}$$

This is inspired by the sequences of abstract states, which consist of two components ($\widehat{t}$ and $\widehat{u}$). Similarly to the principle depicted in Figure 5.1, an edge of the graph must not connect two nodes of the same partition.

$$\mathit{Edges}^{\mathit{detail,prog},C_i} \subseteq (T^{\mathit{detail,prog},C_i} \times U^{\mathit{detail,prog},C_i}) \cup (U^{\mathit{detail,prog},C_i} \times T^{\mathit{detail,prog},C_i}) \tag{6.98}$$

In the same way, a node of partition U must be mapped to an abstract state that describes at least all system states that the abstract states of its predecessor nodes describe. This corresponds to a joining/widening operation. Thus, we refer to all edges from partition T to partition U as *joining/widening edges*. If no actual joining/widening is applied at a joining/widening edge, its target node is mapped to the same abstract states as its source node.

$$\mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{jn/wdn}} = \mathit{Edges}^{\mathit{detail,prog},C_i} \cap (T^{\mathit{detail,prog},C_i} \times U^{\mathit{detail,prog},C_i}) \tag{6.99}$$

$$\begin{gathered}\forall (\mathit{nd}_1, \mathit{nd}_2) \in \mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{jn/wdn}} : \\ \mathit{toState}(\mathit{nd}_1) \sqsubseteq \mathit{toState}(\mathit{nd}_2)\end{gathered} \tag{6.100}$$

Moreover, no node in partition T shall have more than one successor node. Intuitively, it makes no sense to apply multiple different joining/widening operations to the same originating abstract state.

$$\forall \mathit{nd} \in T^{\mathit{detail,prog},C_i} : |\mathit{outEdges}(\mathit{nd}_1)| \leq 1 \tag{6.101}$$

In addition to an abstract state, each node of the graph is also mapped to a (potentially empty) list of basic block end instructions of the considered program that retired on the considered core during the most recent cycle transition. Intuitively, this mapping is later used to assign each node to a particular basic block.

$$\mathit{rtrd}_{\mathit{prog},C_i} : \mathit{Nodes}^{\mathit{detail,prog},C_i} \to \mathit{InstrMemAddrList} \tag{6.102}$$

We obtain the set of basic block end instructions of a program on a particular core by selecting the last instruction of each basic block.

$$\mathit{BBEndInstr}_{\mathit{prog},C_i} = \{\mathit{back}(\mathit{bb}) \mid \mathit{bb} \in \mathit{BasicBlocks}_{\mathit{prog},C_i}\} \tag{6.103}$$

An edge from partition U to partition T is referred to as *cycle transition edge* if its source node is mapped to an empty list of retired basic block end instructions. For a cycle transition edge, we require that the abstract state of the target node is the result of a cycle transition of the abstract state of the source node. Moreover, the target node remembers the list of basic block end instructions that retired during this cycle transition.

$$\begin{aligned} \mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{cycl\text{-}trns}} = \{(\mathit{nd}_1, \mathit{nd}_2) \in \mathit{Edges}^{\mathit{detail,prog},C_i} \cap (U^{\mathit{detail,prog},C_i} \times T^{\mathit{detail,prog},C_i}) \mid \\ \mathit{len}(\mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_1)) = 0\} \end{aligned} \tag{6.104}$$

$$\begin{aligned} &\forall (\mathit{nd}_1, \mathit{nd}_2) \in \mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{cycl\text{-}trns}} : \\ &\quad (\mathit{toState}(\mathit{nd}_1), \mathit{toState}(\mathit{nd}_2)) \in \widehat{\mathit{Cycle}} \land \\ &\quad \mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_2) = \mathit{filter}(\mathit{retires}_{C_i}((\mathit{toState}(\mathit{nd}_1), \mathit{toState}(\mathit{nd}_2))), \\ &\qquad\qquad \lambda \mathit{ins}.\mathit{ins} \in \mathit{BBEndInstr}_{\mathit{prog},C_i}) \end{aligned} \tag{6.105}$$

For a joining/widening edge, we remove the first element from a non-empty list of retired basic block end instructions. This corresponds to passing on the control from one basic block to another.

$$\begin{aligned} &\forall (\mathit{nd}_1, \mathit{nd}_2) \in \mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{jn/wdn}} : \\ &\quad [\mathit{len}(\mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_1)) = 0 \Rightarrow \mathit{len}(\mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_2)) = 0] \land \\ &\quad [\mathit{len}(\mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_1)) > 0 \Rightarrow \mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_2) = \mathit{popFront}(\mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_1))] \end{aligned} \tag{6.106}$$

Edges from partition U to partition T with the source node being mapped to a non-empty list of retired basic block end instructions are referred to as *zero cycle transition edges*. They are not covered by equation (6.105). They correspond to the rare special case that the execution of a basic block is fully overlapped by the execution of one of its predecessor basic blocks. Intuitively, the basic block at the front of the list of the source node has been executed within zero processor cycles. Thus, the target node of a zero cycle transition edge is mapped to the same abstract state and the same list of retired basic block end instructions as the source node.

$$\begin{aligned} \mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{zero\text{-}cycl\text{-}trns}} = \{(\mathit{nd}_1, \mathit{nd}_2) \in \mathit{Edges}^{\mathit{detail,prog},C_i} \cap (U^{\mathit{detail,prog},C_i} \times T^{\mathit{detail,prog},C_i}) \mid \\ \mathit{len}(\mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_1)) > 0\} \end{aligned} \tag{6.107}$$

$$\begin{aligned} &\forall (\mathit{nd}_1, \mathit{nd}_2) \in \mathit{Edges}^{\mathit{detail,prog},C_i}_{\mathit{zero\text{-}cycl\text{-}trns}} : \\ &\quad \mathit{toState}(\mathit{nd}_2) = \mathit{toState}(\mathit{nd}_1) \land \mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_2) = \mathit{rtrd}_{\mathit{prog},C_i}(\mathit{nd}_1) \end{aligned} \tag{6.108}$$

Figure 6.1 shows a part of a detailed graph. For convenience, the graphical representation of a node (nd_a to nd_h) contains the abstract state and the list of retired basic block end instructions assigned to the node. The figure shows the two possible cases for an edge from a node of partition U (⊘) to a node of partition T (◯). Either it is a cycle transition edge (cf. equation (6.104)). In the figure, this is e.g. the case for the edges $(\mathit{nd}_e, \mathit{nd}_f)$ and $(\mathit{nd}_g, \mathit{nd}_h)$. Or it is a zero cycle transition edge because the execution of a basic block (here bb_2) is fully overlapped by the execution of one of its predecessor basic blocks (cf. equation (6.107)). In the figure, this is the case for the edge $(\mathit{nd}_b, \mathit{nd}_c)$. All edges from a node of partition T (◯) to a node of partition U (⊘) are joining/widening edges (cf. equation (6.99)). In the figure, this is e.g. the case for the edges $(\mathit{nd}_a, \mathit{nd}_b)$, $(\mathit{nd}_d, \mathit{nd}_e)$, and $(\mathit{nd}_f, \mathit{nd}_g)$.

We say that a node of the detailed graph is an *out-node* of a particular basic block if the corresponding end instruction is in front position of the node's list of retired basic block end instructions and all successor nodes have a shorter list of retired basic block end instructions.

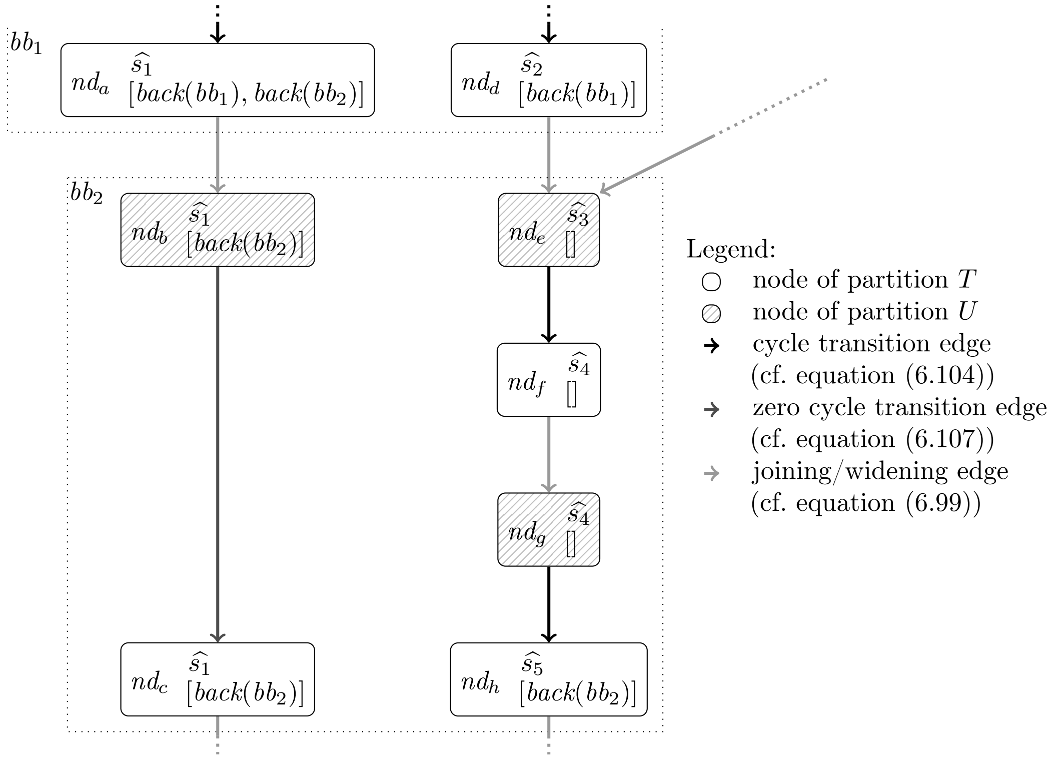

bb_1
nd_a $\widehat{s_1}$ $[back(bb_1), back(bb_2)]$
nd_d $\widehat{s_2}$ $[back(bb_1)]$
bb_2
nd_b $\widehat{s_1}$ $[back(bb_2)]$
nd_e $\widehat{s_3}$ $[]$
nd_f $\widehat{s_4}$ $[]$
nd_g $\widehat{s_4}$ $[]$
nd_c $\widehat{s_1}$ $[back(bb_2)]$
nd_h $\widehat{s_5}$ $[back(bb_2)]$
Legend:
○ node of partition T
▨ node of partition U
→ cycle transition edge (cf. equation (6.104))
→ zero cycle transition edge (cf. equation (6.107))
→ joining/widening edge (cf. equation (6.99))

Figure 6.1.: A part of a detailed graph.

Intuitively, this means that the control was passed on from the basic block in question to one of its successors. In the example of Figure 6.1, nodes nd_a and nd_d are out-nodes of basic block bb_1.

$$\begin{aligned}
&\forall bb \in BasicBlocks_{prog,C_i} :\\
&\quad Nodes^{detail,prog,C_i}_{bb,out} = \{nd \in Nodes^{detail,prog,C_i} \mid\\
&\qquad\qquad [\forall nd' \in succ(nd) : len(rtrd_{prog,C_i}(nd)) > len(rtrd_{prog,C_i}(nd'))] \wedge\\
&\qquad\qquad front(rtrd_{prog,C_i}(nd)) = back(bb)\}
\end{aligned} \tag{6.109}$$

Based on this, each non-start node of the detailed graph is mapped to a basic block in such a way that it can reach one of the basic block's out nodes without traversing another out node of a (potentially different) basic block. Intuitively, this provides a partitioning of the non-start nodes of the detailed graph. In Figure 6.1, we represent this mapping by the dotted boxes corresponding to the basic blocks.

$$\begin{aligned}
&\forall bb \in BasicBlocks_{prog,C_i} :\\
&\quad Nodes^{detail,prog,C_i}_{bb} = \{nd \in Nodes^{detail,prog,C_i} \setminus Nodes^{detail,prog,C_i}_{start} \mid\\
&\qquad\qquad \exists \widehat{p} \in \widehat{SubPaths}^{detail,prog,C_i} :\\
&\qquad\qquad\quad \widehat{p}(0) = nd \wedge \widehat{p}(len(\widehat{p})) \in Nodes^{detail,prog,C_i}_{bb,out} \wedge\\
&\qquad\qquad\quad \forall x \in \mathbb{N}_{<len(\widehat{p})} : \neg\exists bb' \in BasicBlocks_{prog,C_i} :\\
&\qquad\qquad\qquad \widehat{p}(x) \in Nodes^{detail,prog,C_i}_{bb',out}\}
\end{aligned} \tag{6.110}$$

An *in-node* of a basic block is one of its nodes that has a predecessor which either is a start node of the graph or an out-node of a (potentially different) basic block.

$$\begin{aligned}
&\forall bb \in \mathit{BasicBlocks}_{prog,C_i} : \\
&\quad \mathit{Nodes}^{detail,prog,C_i}_{bb,in} = \{nd \in \mathit{Nodes}^{detail,prog,C_i}_{bb} \mid \exists nd' \in \mathit{pred}(nd) : \\
&\qquad\qquad nd' \in \mathit{Nodes}^{detail,prog,C_i}_{start} \lor \\
&\qquad\qquad \exists bb' \in \mathit{BasicBlocks}_{prog,C_i} : \\
&\qquad\qquad\quad nd' \in \mathit{Nodes}^{detail,prog,C_i}_{bb',out}\}
\end{aligned} \tag{6.111}$$

Intuitively, the in-nodes of all basic blocks form the actual fixed point of the micro-architectural analysis. The remaining nodes of the basic blocks are obtained by once more propagating the in-nodes through their respective basic blocks (as already done during the fixed point iteration).

Additionally, the following assumptions shall hold for the start nodes of the detailed graph.

$$\forall nd \in \mathit{Nodes}^{detail,prog,C_i}_{start} : \mathit{len}(\mathit{rtrd}_{prog,C_i}(nd)) = 0 \tag{6.112}$$

$$\mathit{Nodes}^{detail,prog,C_i}_{start} \subseteq T^{detail,prog,C_i} \tag{6.113}$$

This implies that all in-nodes of the basic blocks belong to partition U and all out-nodes of the basic blocks and the start nodes of the graph belong to partition T. Consequently, all edges from start nodes of the graph or out-nodes of a basic block to in-nodes of a basic block are joining/widening edges. As a result, each (optionally zero) cycle transition edge can be mapped to the unique basic block that is assigned to its source and target node. Note that these observations can also be made for the small part of a detailed graph shown in Figure 6.1.

As the graph is the result of a fixed point iteration starting from the start nodes of the graph, it additionally fulfills the following four criteria. The first criterion states that each initial abstract state of the program is assigned to a start node of the detailed graph. Intuitively, this means that every initial abstract state of the program is considered during the fixed point iteration.

$$\forall \widehat{s_i} \in \widehat{\mathit{InitStates}}_{prog,C_i} : \exists nd \in \mathit{Nodes}^{detail,prog,C_i}_{start} : \widehat{s_i} = \mathit{toState}(nd) \tag{6.C3}$$

The second criterion states that any node in partition U that is a starting point for cycle transition edges (cf. equation (6.104)) must have a set of successor nodes that soundly covers all abstract states that can result from a cycle transition of the own abstract state. Intuitively, this means that no result of a cycle transition on abstract states is ever forgotten during the fixed point iteration.

$$\begin{aligned}
&\forall nd \in U^{detail,prog,C_i} : \\
&\quad \mathit{len}(\mathit{rtrd}_{prog,C_i}(nd)) = 0 \Rightarrow \\
&\quad [\forall \widehat{s}' \in \widehat{S} : \\
&\qquad (\mathit{toState}(nd), \widehat{s}') \in \widehat{\mathit{Cycle}} \Rightarrow \\
&\qquad \exists nd' \in \mathit{succ}(nd) : \widehat{s}' = \mathit{toState}(nd')]
\end{aligned} \tag{6.C4}$$

The third criterion states that any node in partition U that is a starting point for zero cycle transition edges (cf. equation (6.107)) also has a successor node. If the execution of a basic block was fully overlapped by the execution of one of its predecessor basic blocks it is important to still pipe the corresponding abstract state through the basic block with a zero cycle transition edge in order to make sure that it is considered in an in-node of one of the successor basic blocks.

Intuitively, this guarantees that we do not lose any abstract states during the fixed point iteration due to a basic block being fully overlapped by one of its predecessor basic blocks.

$$\begin{aligned}&\forall nd \in U^{detail,prog,C_i} :\\ &\quad len(rtrd_{prog,C_i}(nd)) > 0 \Rightarrow\\ &\quad |succ(nd)| = 1\end{aligned} \tag{6.C5}$$

The fourth criterion states that any node in partition T that has no program end instruction at the front of its retired basic block end instructions list must have a successor node (i.e. an outgoing joining/widening edge, cf. equation (6.99)). Intuitively, this means that the successors of any initial abstract state of the program and any result of an abstract cycle transition must be considered during the fixed point iteration. However, if a node has only incoming cycle transition edges that terminated the program it is not required to further follow its successors.

$$\begin{aligned}&\forall nd \in T^{detail,prog,C_i} :\\ &\quad [len(rtrd_{prog,C_i}(nd)) = 0 \vee front(rtrd_{prog,C_i}(nd)) \notin endInstr(prog)] \Rightarrow\\ &\quad |succ(nd)| > 0\end{aligned} \tag{6.C6}$$

The end nodes of the detailed graph are defined as those nodes of partition T that have a program end instruction at the front of their retired basic block end instructions list.

$$\begin{aligned}Nodes^{detail,prog,C_i}_{end} = \{nd \in T^{detail,prog,C_i} \mid\ & len(rtrd_{prog,C_i}(nd)) > 0\ \wedge\\ & front(rtrd_{prog,C_i}(nd)) \in endInstr(prog)\}\end{aligned} \tag{6.114}$$

It follows from our assumptions that each end node is an out-node of a basic block.

$$Nodes^{detail,prog,C_i}_{end} \subseteq \bigcup_{bb \in BasicBlocks_{prog,C_i}} Nodes^{detail,prog,C_i}_{bb,out} \tag{6.115}$$

For cycle transition edges, the event bounding edge weights are identical to the corresponding event bounds of the underlying cycle transitions.

$$\begin{aligned}&\forall (nd_1, nd_2) \in Edges^{detail,prog,C_i}_{cycl\text{-}trns} : \forall E \in Events : \forall BD \in \{UB, LB\}\\ &\widehat{wE^{BD}}((nd_1, nd_2)) = \left|\{(toState(nd_1), toState(nd_2))\} \cap \widehat{E^{BD}}\right|\end{aligned} \tag{6.116}$$

For all other edges, the event bounding edge weights are zero.

$$\begin{aligned}&\forall edg \in (Edges^{detail,prog,C_i} \setminus Edges^{detail,prog,C_i}_{cycl\text{-}trns}) : \forall E \in Events : \forall BD \in \{UB, LB\}\\ &\widehat{wE^{BD}}(edg) = 0\end{aligned} \tag{6.117}$$

It follows from the specification of the detailed graph that its set of relaxed paths provides an overapproximation of $\widehat{ExecRuns}_{prog,C_i}$ and its set of paths provides an overapproximation of $\widehat{ExecRuns}^{term}_{prog,C_i}$. For a formal proof of both statements, we refer to page 284.

$$\bigcup_{\widehat{p} \in \widehat{RelPaths^{detail,prog,C_i}}} \gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \supseteq \widehat{ExecRuns}_{prog,C_i} \tag{6.118}$$

$$\bigcup_{\widehat{p} \in \widehat{Paths^{detail,prog,C_i}}} \gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \supseteq \widehat{ExecRuns}^{term}_{prog,C_i} \tag{6.119}$$

Intuitively, this means that it is safe to calculate event bounds based on the detailed graph. Matthies exploits this in his diploma thesis [Matthies, 2006] in order to calculate event bounds by applying efficient longest path algorithms to a detailed graph. Infeasible paths in the graph can safely be detected by applying system properties that have been lifted to paths through a graph (cf. Section 5.3). It is, however, unclear how to incorporate the lifted versions of cumulative system properties (as e.g. cache persistence properties) into the longest path algorithms without losing their efficiency. The work by Matthies [Matthies, 2006] is limited to loop bounding system properties. Thus, later approaches [Stein, 2010; Cullmann, 2013] resort to an implicit path enumeration (cf. Section 5.4) via ILP as it offers a greater flexibility by being able to encode lifted versions of cumulative system properties as sets of integer linear constraints.

Note that the detailed graph is constructed at cycle granularity (i.e. each edge corresponds to at most one cycle transition on abstract states). Thus, it is intuitively possible to safely use properties lifted to sequences of abstract states in order to detect infeasible paths. The properties lifted to sequences of abstract states might be less pessimistic than the versions that are further lifted to paths through a graph. In this thesis, however, we do not use system properties lifted to sequences of abstract states in order to prune infeasible paths of the detailed graph. Nonetheless, we provide a formal soundness proof for this approach on page 287.

In our prototype implementation, we do not explicitly construct the detailed graph representation. It is only implicitly given by the results of a micro-architectural analysis. Instead, we directly construct less fine-grained graphs (i.e. not at cycle granularity, cf. Section 6.4.4) from the results of a micro-architectural analysis. As long as these graphs subsume the detailed graph, they can be safely used to calculate event bounds. The following subsection defines the corresponding subsumption relation between graphs and points out its implications for the calculation of event bounds.

6.4.3. Calculating Safe Event Bounds on a Graph

In this subsection, we present the calculation of safe event bounds on any graph that *subsumes* the detailed graph. Intuitively, a graph subsumes the detailed graph if it describes at least all sequences of abstract states described by the detailed graph. The formal definition of graph subsumption, however, is a bit more sophisticated.

We formally define the *graph subsumption* relation between two (potentially but not necessarily different) graphs G^A and G^B. To this end, we first define what subsumption between subpaths means. For this definition, we reuse the formal concept of partitionings as introduced in Section 5.3. A subpath $\widehat{p_B}$ subsumes a subpath $\widehat{p_A}$ ($\widehat{p_B} \vDash_{path} \widehat{p_A}$) if there is a partitioning of the edges along $\widehat{p_A}$ such that every partition is safely bounded by an edge of $\widehat{p_B}$. This principle is very similar to the description relation between subpaths of a graph and sequences of abstract states (cf. equation (5.69)).

$$\begin{aligned}
&\forall \widehat{p_A} \in \widehat{SubPaths}^A : \forall \widehat{p_B} \in \widehat{SubPaths}^B : \\
&\quad \widehat{p_B} \vDash_{path} \widehat{p_A} \\
&\quad \Leftrightarrow \exists part \in Partitionings(len(\widehat{p_B}), len(\widehat{p_A})) : \\
&\qquad \forall E \in Events : \\
&\qquad\quad \forall x \in \mathbb{N}_{<len(\widehat{p_B})} : \\
&\qquad\qquad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{wE^{UB}}(\widehat{p_A}, i) \leq \widehat{wE^{UB}}(\widehat{p_B}, x) \wedge \\
&\qquad\qquad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{wE^{LB}}(\widehat{p_A}, i) \geq \widehat{wE^{LB}}(\widehat{p_B}, x)
\end{aligned} \tag{6.120}$$

As a consequence of $\widehat{p_B}$ subsuming $\widehat{p_A}$, $\widehat{p_B}$ describes all the sequences of abstract states that $\widehat{p_A}$ describes. For a formal proof of this statement, we refer to page 288.

$$\widehat{p_B} \vDash_{path} \widehat{p_A} \Rightarrow \gamma^{B}_{path,prog,C_i}(\widehat{p_B}) \supseteq \gamma^{A}_{path,prog,C_i}(\widehat{p_A}) \tag{6.121}$$

We say that a graph G^B subsumes a (potentially but not necessarily different) graph G^A ($G^B \vDash G^A$) if the following four criteria are fulfilled. The first criterion states that every path through G^A has to be subsumed by a path through G^B.

$$\forall \widehat{p_A} \in \widehat{Paths}^A : \exists \widehat{p_B} \in \widehat{Paths}^B : \widehat{p_B} \vDash_{path} \widehat{p_A} \tag{6.C7}$$

For the remaining three criteria, we require that there exists a relation $\vdash_{path}$ between the relaxed paths of G^B and G^A such that all three criteria hold. The statement $\widehat{p_B} \vdash_{path} \widehat{p_A}$ intuitively means that $\widehat{p_B}$ represents $\widehat{p_A}$.

$$\exists \vdash_{path} \subseteq \widehat{RelPaths}^B \times \widehat{RelPaths}^A : (6.C8) \wedge (6.C9) \wedge (6.C10) \tag{6.122}$$

The second criterion states that a relaxed path of graph G^B shall only represent a relaxed path of graph G^A if it also subsumes it.

$$\forall \widehat{p_A} \in \widehat{RelPaths}^A : \forall \widehat{p_B} \in \widehat{RelPaths}^B : \widehat{p_B} \vdash_{path} \widehat{p_A} \Rightarrow \widehat{p_B} \vDash_{path} \widehat{p_A} \tag{6.C8}$$

The third criterion states that any relaxed path of graph G^A with a length of zero shall be represented by a relaxed path of graph G^B with a length of zero.

$$\begin{aligned} &\forall \widehat{p_A} \in \widehat{RelPaths}^A : \\ &\quad len(\widehat{p_A}) = 0 \Rightarrow \\ &\quad \exists \widehat{p_B} \in \widehat{RelPaths}^B : len(\widehat{p_B}) = 0 \wedge \widehat{p_B} \vdash_{path} \widehat{p_A} \end{aligned} \tag{6.C9}$$

The fourth and final criterion states that there shall be a natural number n such that for every relaxed path $\widehat{p_A}$ of G^A that is represented by a relaxed path $\widehat{p_B}$ of G^B and for every possible extension $\widehat{p_A}'$ of $\widehat{p_A}$ by at least n nodes, there shall be another strict extension of $\widehat{p_A}$ that is also a prefix of $\widehat{p_A}'$ and represented by a strict extension of $\widehat{p_B}$.

$$\begin{aligned} &\exists n \in \mathbb{N} : \\ &\quad \forall \widehat{p_A}, \widehat{p_A}' \in \widehat{RelPaths}^A : \forall \widehat{p_B} \in \widehat{RelPaths}^B : \\ &\quad\quad \widehat{p_B} \vdash_{path} \widehat{p_A} \wedge (\widehat{p_A}, \widehat{p_A}') \in \widehat{PrefixOf}^A_{path} \wedge len(\widehat{p_A}') \geq len(\widehat{p_A}) + n \Rightarrow \\ &\quad\quad \exists \widehat{p^{ext}_A} \in \widehat{RelPaths}^A : \exists \widehat{p^{ext}_B} \in \widehat{RelPaths}^B : \\ &\quad\quad\quad (\widehat{p_A}, \widehat{p^{ext}_A}) \in \widehat{StrictPrefixOf}^A_{path} \wedge \\ &\quad\quad\quad (\widehat{p^{ext}_A}, \widehat{p_A}') \in \widehat{PrefixOf}^A_{path} \wedge \\ &\quad\quad\quad (\widehat{p_B}, \widehat{p^{ext}_B}) \in \widehat{StrictPrefixOf}^B_{path} \wedge \\ &\quad\quad\quad \widehat{p^{ext}_B} \vdash_{path} \widehat{p^{ext}_A} \end{aligned} \tag{6.C10}$$

A *feedback node set* [Karp, 1972] of a graph is a subset of its nodes that is chosen in a way that each cycle of the graph contains at least one node of the subset. Let $\mathit{Feedback}^B$ denote a valid choice of feedback nodes for graph G^B. Note that, in contrast to the classical feedback node set problem considered by Karp, we do not require the feedback node set to be minimal or to at most contain a given number of nodes. For our approach, any valid set of feedback nodes will work.

$$\mathit{Feedback}^B \subseteq \mathit{Nodes}^B \tag{6.123}$$

$$\begin{aligned}&\forall \widehat{p} \in \widehat{\mathit{SubPaths}}^B :\\ &\quad [\exists x_1, x_2 \in \mathbb{N}_{\leq len(\widehat{p})} : x_1 \neq x_2 \wedge \widehat{p}(x_1) = \widehat{p}(x_2)] \Rightarrow\\ &\quad \exists x \in \mathbb{N}_{\leq len(\widehat{p})} : \widehat{p}(x) \in \mathit{Feedback}^B\end{aligned} \tag{6.124}$$

The set $\widehat{\mathit{FeedPaths}}^B$ contains the relaxed paths of graph G^B that end in an end node of the graph or in a member of $\mathit{Feedback}^B$.

$$\widehat{\mathit{FeedPaths}}^B = \{\widehat{p} \in \widehat{\mathit{RelPaths}}^B \mid \widehat{p}(len(\widehat{p})) \in \mathit{Nodes}^B_{end} \cup \mathit{Feedback}^B\} \tag{6.125}$$

$\widehat{\mathit{LessFeedPaths}}^B_{prog,C_i}$ denotes the subset of these relaxed paths for which the program properties in Prop_{prog,C_i} lifted to paths through a graph hold.

$$\widehat{\mathit{LessFeedPaths}}^B_{prog,C_i} = \{\widehat{p} \in \widehat{\mathit{FeedPaths}}^B \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P^{path}_k}(\widehat{p})\} \tag{6.126}$$

Based on $\widehat{\mathit{LessFeedPaths}}^B_{prog,C_i}$, we specify upper and lower event bounds.

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad \widehat{\mathit{Maximum}}^{B,path}_{prog,C_i,E} = \max_{\widehat{p} \in \widehat{\mathit{LessFeedPaths}}^B_{prog,C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{\mathit{wEvent}}^{UB}_{prog,C_i,E}(\widehat{p}, x)\end{aligned} \tag{6.127}$$

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad \widehat{\mathit{Minimum}}^{B,path}_{prog,C_i,E} = \min_{\widehat{p} \in \widehat{\mathit{LessFeedPaths}}^B_{prog,C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p}, x)\end{aligned} \tag{6.128}$$

If graph G^B subsumes the detailed graph and one of the event bounds based on it has a defined value, this value is guaranteed to safely bound the actual number of event occurrences of the concrete system. This is formally expressed by the two following statements. For a formal proof of these soundness statements, we refer to page 289.

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad [G^B \models G^{detail,prog,C_i} \wedge \widehat{\mathit{Maximum}}^{B,path}_{prog,C_i,E} \in \mathbb{N}] \Rightarrow\\ &\quad \forall t \in \mathit{ExecRuns}_{prog,C_i} : \mathit{numEvOccur}(prog, C_i, t, E) \leq \widehat{\mathit{Maximum}}^{B,path}_{prog,C_i,E}\end{aligned} \tag{6.129}$$

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad [G^B \models G^{detail,prog,C_i} \wedge \widehat{\mathit{Minimum}}^{B,path}_{prog,C_i,E} \in \mathbb{N}] \Rightarrow\\ &\quad \forall t \in \mathit{ExecRuns}^{min\text{-}relev}_{prog,C_i,E} : \mathit{numEvOccur}(prog, C_i, t, E) \geq \widehat{\mathit{Minimum}}^{B,path}_{prog,C_i,E}\end{aligned} \tag{6.130}$$

We only consider graphs G^B that contain a finite number of nodes.

$$\exists n \in \mathbb{N} : \left|\mathit{Nodes}^B\right| = n \tag{6.131}$$

Thus, the value of $\widehat{Minimum}^{B,path}_{prog,C_i,E}$ is always defined for a program *prog* that is actually executed on core C_i (i.e. $ExecRuns_{prog,C_i} \neq \emptyset$). The value of $\widehat{Maximum}^{B,path}_{prog,C_i,E}$, in contrast, can be undefined if the number of occurrences of event E cannot be upper-bounded at all or if graph G^B and the lifted versions of the properties in $Prop_{prog,C_i}$ are not sufficient to prove an upper bound.

In addition to the terminated program execution runs, the exact lower bounds $Minimum_{prog,C_i,E}$ only argue about those diverging execution run prefixes that might be continued indefinitely without producing another event E (cf. $ExecRuns^{min\text{-}relev}_{prog,C_i,E}$ as defined in equation (6.29)). Thus, the lower bounds $\widehat{Minimum}^{B,path}_{prog,C_i,E}$ might be overly pessimistic in some cases as $Feedback^B$ might also contain some feedback nodes that only belong to cycles *cyc* in the graph that are guaranteed to produce further events E (i.e. $\{edg \in cyc \mid \widehat{wEvent}^{LB}_{prog,C_i,E}(edg) > 0\} \neq \emptyset$).

To overcome this problem, we specify a variant $\widehat{Minimum}^{B,path'}_{prog,C_i,E}$ of a lower event bound on the graph G^B. It makes use of a more particular set $Feedback^B_E$ of feedback nodes, which is needed per event E for which a potentially more precise lower event bound is desired. $Feedback^B_E$ only has to contain one node per cycle of graph G^B that does not guarantee any further events E.

$$\forall E \in Events : Feedback^B_E \subseteq Nodes^B \tag{6.132}$$

$$\begin{aligned}&\forall E \in Events : \forall \widehat{p} \in \widehat{SubPaths}^B :\\ &\quad [(\forall x \in \mathbb{N}_{<len(\widehat{p})} : \widehat{wEvent}^{LB}_{prog,C_i,E}(\widehat{p}, x) = 0) \wedge\\ &\quad\quad (\exists x_1, x_2 \in \mathbb{N}_{\leq len(\widehat{p})} : x_1 \neq x_2 \wedge \widehat{p}(x_1) = \widehat{p}(x_2))] \Rightarrow\\ &\quad \exists x \in \mathbb{N}_{\leq len(\widehat{p})} : \widehat{p}(x) \in Feedback^B_E\end{aligned} \tag{6.133}$$

The specification of $\widehat{Minimum}^{B,path'}_{prog,C_i,E}$ is analogous to the specification of $\widehat{Minimum}^{B,path}_{prog,C_i,E}$. Thus, we only present the corresponding formulae without further explanation.

$$\begin{aligned}&\forall E \in Events :\\ &\quad \widehat{FeedPaths}^B_E = \{\widehat{p} \in \widehat{RelPaths}^B \mid \widehat{p}(len(\widehat{p})) \in Nodes^B_{end} \cup Feedback^B_E\}\end{aligned} \tag{6.134}$$

$$\begin{aligned}&\forall E \in Events :\\ &\widehat{LessFeedPaths}^B_{prog,C_i,E} = \{\widehat{p} \in \widehat{FeedPaths}^B_E \mid \forall P_k \in Prop_{prog,C_i} : \widehat{P}^{path}_k(\widehat{p})\}\end{aligned} \tag{6.135}$$

$$\begin{aligned}&\forall E \in Events :\\ &\widehat{Minimum}^{B,path'}_{prog,C_i,E} = \min_{\widehat{p} \in \widehat{LessFeedPaths}^B_{prog,C_i,E}} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wEvent}^{LB}_{prog,C_i,E}(\widehat{p}, x)\end{aligned} \tag{6.136}$$

$\widehat{Minimum}^{B,path'}_{prog,C_i,E}$ provides the same soundness guarantee as $\widehat{Minimum}^{B,path}_{prog,C_i,E}$. For a formal proof of this soundness statement, we refer to page 297.

$$\begin{aligned}&\forall E \in Events :\\ &[G^B \vDash G^{detail,prog,C_i} \wedge \widehat{Minimum}^{B,path'}_{prog,C_i,E} \in \mathbb{N}] \Rightarrow\\ &\forall t \in ExecRuns^{min\text{-}relev}_{prog,C_i,E} : numEvOccur(prog, C_i, t, E) \geq \widehat{Minimum}^{B,path'}_{prog,C_i,E}\end{aligned} \tag{6.137}$$

In contrast to $\widehat{Minimum}^{B,path}_{prog,C_i,E}$, the value of $\widehat{Minimum}^{B,path'}_{prog,C_i,E}$ might be undefined even for a program *prog* that is actually executed on core C_i (i.e. $ExecRuns_{prog,C_i} \neq \emptyset$). This means that it is guaranteed that the program diverges and that it never stops producing occurrences of the event E under any circumstance.

The downside of the potentially increased precision of $\widehat{Minimum}^{B,path'}_{prog,C_i,E}$ is that the corresponding set of feedback nodes of the graph has to be determined independently per event E for which a lower event bound is desired. Note that it is not possible to further improve the precision of the upper event bounds in a similar way.

The respective feedback nodes that the relaxed paths considered in this subsection might additionally end in can also be seen as additional nodes that we insert into the set of end nodes of a graph. In this way, we consider again all paths from the start nodes to the end nodes of a graph and, as a consequence, can directly apply implicit path enumeration. Thus, intuitively, this chapter demonstrates that—in general—a set of feedback nodes of the graph added to the end nodes of the graph is sufficient in order to calculate sound event bounds based on all paths through the graph. In Section 6.4.7, we argue that this consideration of the feedback nodes is not necessary in case the program under analysis is guaranteed to terminate.

Criteria (6.C7), (6.C8), (6.C9), and (6.C10) imply that the *graph subsumption* relation *is reflexive* (i.e. every graph subsumes itself). Thus, in particular, the detailed graph subsumes itself. In this subsection, we have shown that any graph subsuming the detailed graph can safely be used for the calculation of event bounds. As a consequence, the technique presented in this subsection (i.e. adding the feedback nodes to the graph's set of end nodes and subsequently calculating the event bounds) can—in principle—also be directly applied to the detailed graph.

In order to make the analysis tractable (in terms of analysis runtime and memory consumption), however, it is common to consider graphs that are less fine-grained than the detailed graph. Thus, the next subsection presents more coarse graphs which subsume the detailed graph.

6.4.4. Graphs at the Granularity of Basic Blocks

The detailed graph is constructed at cycle granularity. This means that no edge describes a sequence of more than one abstract cycles transition. In this subsection, we present the construction of graphs at basic-block granularity based on the detailed graph. Basic-block granularity means that no edge argues about more than one basic block execution. The resulting graphs are typically significantly more coarse than the detailed graph. Our construction implies that the resulting graphs subsume the detailed graph. Thus, each resulting graph is suitable for a calculation of event bounds based on the feedback node set approach as presented in Section 6.4.3.

Note that the graph construction presented in this section is only conceptually based on the detailed graph representation. Thus, the detailed graph does not have to be explicitly constructed. For an implementation of the presented graph construction, it is sufficient if the detailed graph is implicitly given in the form of the results of the micro-architectural analysis. Nonetheless, for a concise description, we specify the graph construction based on the helper machinery introduced while defining the detailed graph representation (cf. Section 6.4.2).

An edge of a graph at basic-block granularity typically subsumes multiple subpaths within a basic block of the detailed graph. In order to argue about this in a concise way, we formally define the sets of *subpaths within each basic block of the detailed graph.* A subpath of the detailed graph is considered to be within a particular basic block if all nodes on the subpath belong to the basic block and at most the last node on the subpath is an out-node of the basic block.

$$\begin{aligned}
&\forall bb \in BasicBlocks_{prog,C_i} : \\
&\quad \widehat{SubPaths}^{detail,prog,C_i}_{bb} = \{\widehat{p} \in \widehat{SubPaths}^{detail,prog,C_i} \mid \\
&\qquad\qquad [\forall x \in \mathbb{N}_{\leq len(\widehat{p})} : \widehat{p}(x) \in Nodes^{detail,prog,C_i}_{bb}] \wedge \\
&\qquad\qquad [\forall x \in \mathbb{N}_{< len(\widehat{p})} : \widehat{p}(x) \notin Nodes^{detail,prog,C_i}_{bb,out}]\}
\end{aligned} \tag{6.138}$$

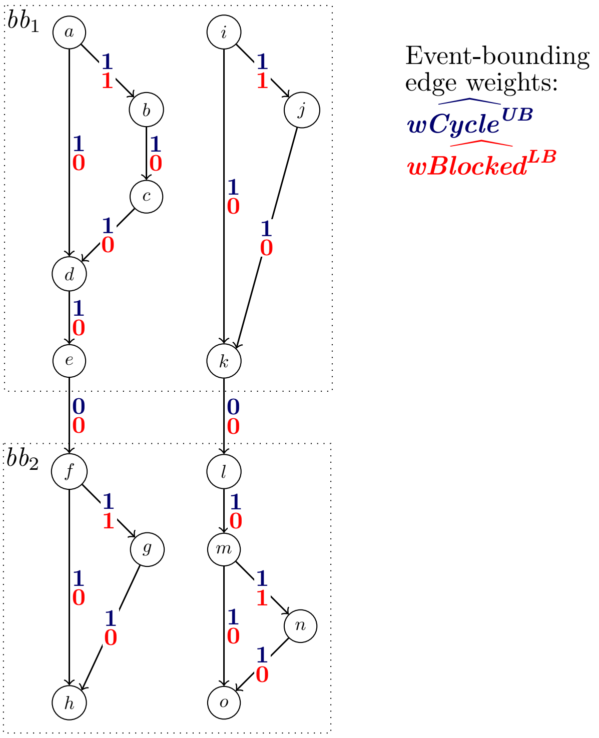


Figure 6.2.: Detailed graph of a simple example program. The edge weights upper-bound the number of **processor cycles** and lower-bound the number of **cycles blocked at the shared bus**.

Before beginning the construction of graphs at basic-block granularity, we state two assumptions about the detailed graph. First, the detailed graph is assumed to be of finite size.

$$\exists n \in \mathbb{N} : \left| Nodes^{detail,prog,C_i} \right| = n \tag{6.139}$$

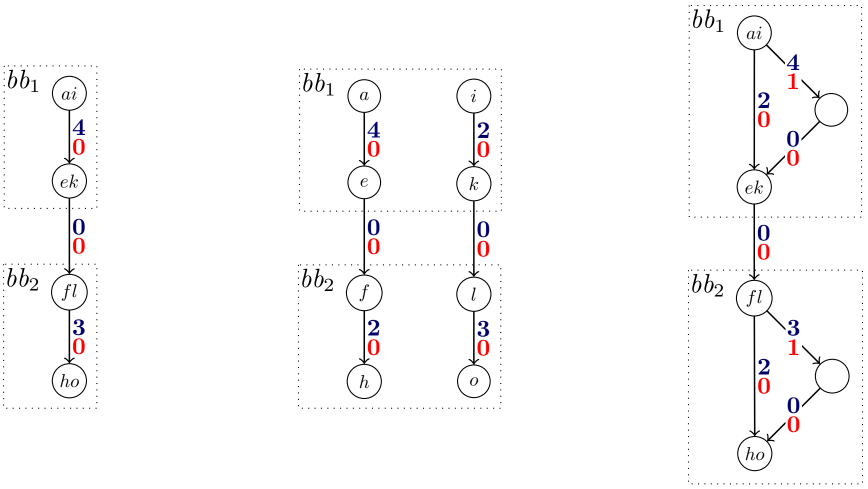
Additionally, we assume that there are no subpaths of unbounded length within any basic block of the detailed graph.

$$\begin{aligned} &\forall bb \in BasicBlocks_{prog,C_i} : \\ &\quad \exists n \in \mathbb{N} : \\ &\qquad \forall \widehat{p} \in \widehat{SubPaths}^{detail,prog,C_i}_{bb} : len(\widehat{p}) \leq n \end{aligned} \tag{6.140}$$

Note that, in particular, the second assumption does not necessarily hold for systems with an unfair bus arbitration policy (e.g. priority-based bus arbitration). We refer to the implementation part of this thesis for a discussion on how to also support such systems (cf. Section 9.2).

Figure 6.2 depicts the detailed graph of a simple example program. We use it as a running example throughout this subsection in order to explain the principles of our construction of graphs at basic-block granularity. For simplicity, the example program only consists of two basic blocks. For the same reason, we also omitted the start nodes of the detailed graph as well as joining/widening edges (cf. Section 6.4.2) within the basic blocks.

We consider the example of Figure 6.2 in the context of WCET analysis. Thus, we would like to obtain an upper bound on the overall number of processor cycles it takes to execute basic block bb_1 followed by basic block bb_2. This bound is referred to as WCET bound in the following.



(a) Every basic block is represented by one edge.

(b) Every connected pair of in- and out-node per basic block is represented by one edge.

(c) Each amount of blocked cycles per basic block is represented by one edge.

Figure 6.3.: Three graphs at basic-block granularity. Each subsumes the detailed graph of Figure 6.2. Their degrees of detail differ significantly.

Note that every edge of the graph is annotated with an upper bound on the number of processor cycles ($\widehat{wCycle}^{UB}$, colored in blue). Finding a WCET bound corresponds to finding a path from an in-node of basic block bb_1 (i.e. node a or i) to an out-node of basic block bb_2 (i.e. node h or o) such that the sum over the blue numbers along the path is maximized. In this way, we obtain a WCET bound of six processor cycles.

Moreover, the edges of the graph are annotated with lower bounds on the numbers of cycles blocked at the shared bus ($\widehat{wBlocked}^{LB}$, colored in red). Note that the path from node a to node h which dominates the WCET bound only argues about concrete traces which are blocked for at least two cycles at the shared bus. If we additionally know that the concrete system cannot suffer from blocked cycles at the shared bus (e.g. because the concurrent processor cores do not access the shared bus), we can safely exclude all paths which guarantee a positive number of blocked cycles. Thus, assuming the absence of bus blocking, we obtain a WCET bound of three processor cycles.

Figure 6.3 presents three graphs at basic block granularity. Each of them subsumes the detailed graph of Figure 6.2. The degrees of detail of the three graphs differ significantly.

The most simple version (Figure 6.3a) of a graph at basic block granularity represents each basic block by one edge. Node ai of this graph represents the in-nodes a and i of bb_1 in the detailed graph. In the same way, node ek of this graph represents the out-nodes e and k of bb_1 in the detailed graph. The weights of the edge between nodes ai and ek are chosen in a way that it subsumes each path from an in-node to an out-node of bb_1 in the detailed graph. Thus, this edge describes every possible sub sequence of a concrete trace corresponding to the execution of basic block bb_1. The same principle holds for basic block bb_2 and the edge between nodes fl and ho.

The principle of representing every basic block by a single edge has been widely used in WCET analysis [Li and Malik, 1995; Engblom and Ermedahl, 2000]. However, it can lead to a significant loss of precision. For our simple example, it leads to a WCET bound of seven processor cycles (instead of six obtained by the detailed graph). Moreover, assuming the absence of bus blocking does not allow us to further reduce the WCET bound.

The detailed graph is aware that the worst cases of both basic blocks (four cycles for bb_1 and three cycles for bb_2) can never happen within the same concrete trace. The graph in Figure 6.3a, in contrast, has only a single pair of in-node and out-node per basic block. Thus, it pessimistically assumes that the worst cases of both basic blocks can happen within the same concrete trace, which leads to an increased WCET bound. We can overcome this problem by distinguishing the in- and out-nodes of the basic blocks in the same way as in the detailed graph. Thus, we end up in one edge per connected pair of in- and out-node per basic block. This principle is referred to as *full node-sensitivity at basic block boundaries.* The corresponding graph is depicted in Figure 6.3b. It results in a WCET bound of six processor cycles, which is as precise as in the detailed graph. However, this graph still cannot provide an improved WCET bound by assuming the absence of bus blocking.

We can effectively use the knowledge about the absence of bus blocking to improve the WCET bound if each possible amount of blocked cycles within a basic block is represented by its own edge. This principle is referred to as *full edge-weight-sensitivity* of the blocked cycles. A corresponding graph is depicted in Figure 6.3c. Assuming the absence of bus blocking, it results in a WCET bound of four processor cycles. The additional dummy nodes in Figure 6.3c (i.e. those that are not labeled) are needed because our graph formalism does not allow multiple directed edges with the same source node and target node.

Note that the principles presented in Figure 6.3b and Figure 6.3c can also be combined. For our simple example, this would result in WCET bounds as precise as those obtained by the detailed graph. Due to the simplicity of our example, however, the resulting graph would not be much smaller than the detailed graph. For larger detailed graphs, the combination of both principles typically still leads to a significant reduction in graph size compared to the detailed graph.

Further note that the principles presented in Figure 6.3b and Figure 6.3c are only the extreme cases of two *orthogonal axes of sensitivity* in the construction of graphs at basic-block granularity. Figure 6.3a combines the opposite extreme cases of these axes: node-insensitivity at basic block boundaries and edge-weight-insensitivity of the blocked cycles. In the following, we formally define these two axes of sensitivity, together with a third one.

Node-Sensitivity at Basic Block Boundaries We formalize a particular node-sensitivity at basic block boundaries as a set of subsets of the nodes of the detailed graph. This set of subsets has to be chosen in a way that it covers the in-nodes and out-nodes of all basic blocks as well as the start nodes of the detailed graph. This is reflected in the following definition of the set $NodeSens_{bndr}^{detail,prog,C_i}$ of all possible node-sensitivities at basic block boundaries.

$$\begin{aligned} NodeSens_{bndr}^{detail,prog,C_i} = \{ & Subsets \subseteq \mathcal{P}(Nodes^{detail,prog,C_i}) \mid \\ & \bigcup Subsets \supseteq \bigcup_{bb \in BasicBlocks_{prog,C_i}} Nodes_{bb,in}^{detail,prog,C_i} \wedge \\ & \bigcup Subsets \supseteq \bigcup_{bb \in BasicBlocks_{prog,C_i}} Nodes_{bb,out}^{detail,prog,C_i} \wedge \\ & \bigcup Subsets \supseteq Nodes_{start}^{detail,prog,C_i} \} \end{aligned} \tag{6.141}$$

The node sensitivity $nodeSens_{bndr} \in NodeSens_{bndr}^{detail,prog,C_i}$ specifies which in-nodes (out-nodes) of a basic block of the detailed graph are represented by a common in-node (out-node) of that basic block in the constructed graph G^{constr}. During the construction of G^{constr}, each non-empty intersection of a member of $nodeSens_{bndr}$ with the set of in-nodes (out-nodes) of a basic block of the detailed graph corresponds to an in-node (out-node) of that basic block in G^{constr}. Analogously, each non-empty intersection of a member of $nodeSens_{bndr}$ with the set of start nodes of the detailed graph corresponds to a start node of G^{constr}. This principle is demonstrated by Algorithm 6.1.

Algorithm 6.1 : Determination of in-, out-, and start nodes of the constructed graph

```
Data : $nodeSens_{bndr} \in NodeSens^{detail,prog,C_i}_{bndr}$, constituents of $G^{detail,prog,C_i}$
Result : in-, out-, and start nodes of $G^{constr}$
begin
    for $bb \in BasicBlocks_{prog,C_i}$ do
        $Nodes^{constr}_{bb,in} \longleftarrow \emptyset$
        for $subset \in nodeSens_{bndr}$ do
            $temp \longleftarrow subset \cap Nodes^{detail,prog,C_i}_{bb,in}$
            if $temp \neq \emptyset$ then
                $Nodes^{constr}_{bb,in} \longleftarrow Nodes^{constr}_{bb,in} \cup \{temp\}$
        $Nodes^{constr}_{bb,out} \longleftarrow \emptyset$
        for $subset \in nodeSens_{bndr}$ do
            $temp \longleftarrow subset \cap Nodes^{detail,prog,C_i}_{bb,out}$
            if $temp \neq \emptyset$ then
                $Nodes^{constr}_{bb,out} \longleftarrow Nodes^{constr}_{bb,out} \cup \{temp\}$
    $Nodes^{constr}_{start} \longleftarrow \emptyset$
    for $subset \in nodeSens_{bndr}$ do
        $temp \longleftarrow subset \cap Nodes^{detail,prog,C_i}_{start}$
        if $temp \neq \emptyset$ then
            $Nodes^{constr}_{start} \longleftarrow Nodes^{constr}_{start} \cup \{temp\}$
```

As an example, consider the extreme case of *node-insensitivity at basic block boundaries*. Intuitively, this means that each basic block of the constructed graph G^{constr} only has a single in-node and a single out-node (cf. Figures 6.3a and 6.3c). Classical approaches to WCET analysis [Li and Malik, 1995; Engblom and Ermedahl, 2000] typically assumed node-insensitivity at basic block boundaries. In our formalization of node-sensitivity at basic block boundaries, this can be expressed by a set containing the set of all nodes of the detailed graph.

$$\{Nodes^{detail,prog,C_i}\} \in NodeSens^{detail,prog,C_i}_{bndr} \tag{6.142}$$

The opposite extreme case is *full node-sensitivity at basic block boundaries*. Intuitively, this means that each of the in- and out-nodes of the detailed graph has a dedicated representative in the constructed graph G^{constr} (cf. Figure 6.3b). It is used to improve the precision of WCET bounds for processors with complex pipelines at the cost of increased graphs/analysis time. In literature, it is also referred to as *prediction-file-based* approach [Cullmann, 2013; Maksoud and Reineke, 2014; Maksoud, 2015] respectively *state-sensitive graph* [Jacobs et al., 2015]. In our formalization of node-sensitivity at basic block boundaries, this can be expressed by partitioning the set of all nodes of the detailed graph into subsets of size one.

$$\{X \subseteq Nodes^{detail,prog,C_i} \mid |X| = 1\} \in NodeSens^{detail,prog,C_i}_{bndr} \tag{6.143}$$

Note that—between these extreme cases—there are various other degrees of node-sensitivity at basic block boundaries. The *context-sensitivity at basic block boundaries* is a prominent example [Theiling et al., 2000; Theiling, 2002]. It features one in-node (out-node) per basic block and analysis context. Thus, each in-node (out-node) of the constructed graph represents all in-nodes (out-nodes) of the detailed graph belonging to a particular basic block and analysis context.

Node-Sensitivity inside of Basic Blocks For certain analysis approaches, it is desired that some of the inner nodes of the basic blocks in the detailed graph are also represented by nodes in the constructed graph. This is e.g. useful for a cache-analysis via implicit path enumeration [Li et al., 1995, 1996]. In the future, this may also be useful for approaches which try to distribute a budget of shared-bus interference between the shared-bus accesses of a program in a way that the WCET bound is maximized. Existing approaches to this distribution [Nagar and Srikant, 2014, 2016] inherently rely on hardware platforms without timing anomalies [Lundqvist and Stenstrom, 1999] and, thus, are not even applicable to relatively simple real-world hardware platforms [Hahn et al., 2016a].

Intuitively, an increased node-sensitivity inside of basic blocks means that we select some of the inner nodes of a basic block in the detailed graph to be represented by a common node in the constructed graph. We can formally describe a particular node-sensitivity inside of basic blocks as a set of subsets of the nodes of the detailed graph. This is reflected in the following definition of the set $NodeSens_{insd}^{detail,prog,C_i}$ of all possible node-sensitivities inside of basic blocks.

$$NodeSens_{insd}^{detail,prog,C_i} = \{Subsets \subseteq \mathcal{P}(Nodes^{detail,prog,C_i})\} \tag{6.144}$$

The node sensitivity $nodeSens_{insd} \in NodeSens_{insd}^{detail,prog,C_i}$ specifies which inner nodes of a basic block of the detailed graph are represented by a common inner node of that basic block in the constructed graph G^{constr}. During the construction of G^{constr}, each non-empty intersection of a member of $nodeSens_{insd}$ with the set of inner nodes of a basic block of the detailed graph corresponds to an inner node of that basic block in G^{constr}. This principle is demonstrated by Algorithm 6.2.

Algorithm 6.2 : Determination of the inner nodes of the basic blocks for the constructed graph

```
Data : $nodeSens_{insd} \in NodeSens_{insd}^{detail,prog,C_i}$, constituents of $G^{detail,prog,C_i}$
Result : inner nodes of the basic blocks of $G^{constr}$
begin
    for $bb \in BasicBlocks_{prog,C_i}$ do
        $Nodes_{bb,inner}^{constr} \longleftarrow \emptyset$
        for $subset \in nodeSens_{insd}$ do
            $temp \longleftarrow subset \cap [Nodes_{bb}^{detail,prog,C_i} \setminus (Nodes_{bb,in}^{detail,prog,C_i} \cup Nodes_{bb,out}^{detail,prog,C_i})]$
            if $temp \neq \emptyset$ then
                $Nodes_{bb,inner}^{constr} \longleftarrow Nodes_{bb,inner}^{constr} \cup \{temp\}$
```

As an example, consider the extreme case of *node-insensitivity inside of basic blocks.* Intuitively, this means that no basic block of the constructed graph G^{constr} has inner nodes representing inner nodes of the basic block in the detailed graph (cf. Figures 6.3a, 6.3b, and 6.3c). Note that the dummy nodes in Figure 6.3c are only needed to increase the edge-weight-sensitivity and, thus, do not represent any inner nodes of the detailed graph of Figure 6.2. Classical approaches to WCET analysis [Li and Malik, 1995; Engblom and Ermedahl, 2000] typically assumed node-insensitivity inside of basic blocks. In our formalization of node-sensitivity inside of basic blocks, this can be expressed by an empty set (i.e. none of the inner nodes of the detailed graph is represented by an inner node of the constructed graph).

$$\emptyset \in NodeSens_{insd}^{detail,prog,C_i} \tag{6.145}$$

The opposite extreme case is *full node-sensitivity inside of basic blocks*. Intuitively, this means that each inner node of the detailed graph has a dedicated representative in the constructed graph G^{constr}. In our formalization of node-sensitivity inside of basic blocks, this can be expressed by partitioning the set of all nodes of the detailed graph into subsets of size one.

$$\{X \subseteq Nodes^{detail,prog,C_i} \mid |X| = 1\} \in NodeSens_{insd}^{detail,prog,C_i} \tag{6.146}$$

Note that—between these extreme cases—there are various other degrees of node-sensitivity inside of basic blocks. The cache analysis based on implicit path enumeration [Li et al., 1995, 1996] e.g. introduces additional nodes representing the nodes of the detailed graph at which accesses to the cache are initiated. Subsequently, it checks the following property: for each path through the resulting graph there is a corresponding path through the cache state transition graph rectifying the respective numbers of cache hits and misses for the different memory blocks accessed along the former path. In order to check this property during implicit path enumeration, the cache state transition graph is modeled by additional integer variables. This approach, however, turned out to not scale to caches and programs of realistic size.

Further note that the combined extreme case of *full node-sensitivity at basic block boundaries and inside of basic blocks* leads to a constructed graph that has a dedicated representative node for every node of the detailed graph. Thus, the constructed graph is *isomorphic* to the detailed graph. (Independently of the chosen edge-weight-sensitivity, no dummy nodes will be needed in this case.)

Edge-Weight-Sensitivity Intuitively, an edge-weight-sensitivity uses the edge weights in order to decide which subpaths of the detailed graph are represented by a common edge of the constructed graph. We specify this sensitivity for a particular event-bounding edge weight (e.g. $\widehat{wBlocked}^{LB}$, cf. Figure 6.2) by a *covering of the natural numbers*. A covering of the natural numbers is a set of subsets of the natural numbers chosen in a way that each natural number is a member of at least one of the subsets. The set $Coverings_{\mathbb{N}}$ formalizes the space of all possible coverings of the natural numbers.

$$Coverings_{\mathbb{N}} = \{cov \subseteq \mathcal{P}(\mathbb{N}) \mid \bigcup cov \supseteq \mathbb{N}\} \tag{6.147}$$

We bind a covering of the natural numbers to a particular system event and to a particular bound direction (i.e. upper or lower bound) by an *event bound sensitivity tuple*. The space of all such tuples is defined as follows.

$$EventBoundSensTuples = Events \times \{UB, LB\} \times Coverings_{\mathbb{N}} \tag{6.148}$$

Intuitively, we can combine different weight sensitivities for different pairs of system event and bound direction. Thus, an *edge-weight-sensitivity* is a set of event bound sensitivity tuples. However, it must be guaranteed that the same pair of system event and bound direction is not assigned multiple different coverings. The resulting space of possible edge weight sensitivities is defined as follows.

$$\begin{aligned} EdgeWeightSens = \{\, & Tuples \subseteq EventBoundSensTuples \mid \\ & \neg\exists (E_1, BD_1, cov_1), (E_2, BD_2, cov_2) \in Tuples : \\ & E_1 = E_2 \wedge BD_1 = BD_2 \wedge cov_1 \neq cov_2 \} \end{aligned} \tag{6.149}$$

Considering a particular edge-weight-sensitivity $edgeWeightSens \in EdgeWeightSens$, we finally define the edges, event-bounding edge weights, and potentially necessary dummy nodes of the constructed graph G^{constr}. Essentially, for every pair of nodes in the constructed graph, the set of all subpaths between the represented nodes in the detailed graph is transformed into potentially multiple sets of subpaths in a way that respects the given edge-weight-sensitivity. Subsequently, every resulting subset is represented by a dedicated edge in the constructed graph. In case this would result in multiple edges with identical source and target nodes, dummy nodes have to be introduced. Algorithm 6.3 presents the details of this procedure.

As an example, consider the extreme case of *edge-weight-insensitivity* (cf. Figures 6.3a and 6.3b). Classical approaches to WCET analysis [Li and Malik, 1995; Engblom and Ermedahl, 2000] are typically edge-weight-insensitive. In our formalization of edge-weight-sensitivity, this can be expressed by an empty set (i.e. no event bound sensitivity tuple is specified).

$$\emptyset \in \mathit{EdgeWeightSens} \tag{6.150}$$

Furthermore, consider the case of *full edge-weight-sensitivity* for a particular event-bounding edge weight. Figure 6.3c shows full edge-weight-sensitivity for the lower bound on the number of blocked cycles. In our formalization of edge-weight-sensitivity, this particular sensitivity can be expressed by the following set containing a single event bound sensitivity tuple.

$$\{(\mathit{Blocked}, \mathit{LB}, \{X \subseteq \mathbb{N} \mid |X| = 1\})\} \in \mathit{EdgeWeightSens} \tag{6.151}$$

A practical example for the importance of a sufficient edge-weight-sensitivity is the consideration of cache-persistence constraints [Stein, 2010; Cullmann, 2013]. Intuitively, such constraints state that a particular path can be excluded if it only describes concrete traces which assume more than one cache miss for a memory block at a point of the program where this block is persistent (i.e. it cannot miss the cache more than once). In order to effectively use these constraints, however, we must represent the cases of cache hit and cache miss of a persistent memory block by different edges in the graph. Previous work on cache persistence was mostly focused on the detection of persistent memory blocks. It did not discuss the details of the graph construction. We argue that the edge-weight-sensitivity has a significant impact on the effectiveness of cache-persistence constraints and, thus, is worthwhile discussing.

In the same way, a sufficient edge-weight-sensitivity is of utmost importance for the effectiveness of constraints that upper-bound the number of cycles blocked at a shared bus. For example, consider the difference between the WCET bounds obtained from the graphs in Figures 6.3a and 6.3c under the additional constraint that there must be no blocked cycles. Thus, the edge-weight-sensitivity is an important parameter in our case study on multi-core processors with shared buses.

Note that we formalized a relatively simple space of possible edge-weight-sensitivities. There are more complex forms of edge-weight-sensitivity that exceed this space, e.g. different degrees of sensitivity for a particular event-bounding edge weight depending on the considered pair of nodes in the constructed graph or the value of a different event-bounding edge weight. Note that different spaces of edge-weight-sensitivities can be supported by adapting the second loop of Algorithm 6.3. A detailed discussion of this, however, is beyond the scope of this thesis.

A Parametric Graph at Basic-Block Granularity Finally, we define the end nodes of the constructed graph G^{constr} as all nodes that represent an end node of the detailed graph $G^{detail,prog,C_i}$.

$$\mathit{Nodes}^{constr}_{end} = \{nd \in \mathit{Nodes}^{constr} \mid nd \notin \mathit{DummyNodes} \wedge \exists nd' \in \mathit{Nodes}^{detail,prog,C_i}_{end} : nd' \in nd\} \tag{6.152}$$

Note that the presented construction of graph G^{constr} is still of conceptual nature—in particular because Algorithm 6.3 iterates over the set *Events* of all possible events of the concrete system (cf. equation (5.6)). An actual implementation would only consider the pairs of events and bound directions which are actually used during the bound calculation or in the lifted properties. Nonetheless, assuming a finite set *Events*, the presented construction of graph G^{constr} is guaranteed to terminate. This can be proved with the help of assumptions (6.139) and (6.140). However, be aware that the presented pseudo-code algorithms mostly serve as a specification and, thus, are by no means optimized with respect to runtime.

Algorithm 6.3 : Determination of the edges, event-bounding edge weights, and dummy nodes for the constructed graph

```
Data : $edgeWeightSens \in EdgeWeightSens$, constituents of $G^{detail,prog,C_i}$, nodes of $G^{constr}$
Result : edges, edge weights, and dummy nodes of $G^{constr}$
begin
    $Nodes^{constr} \longleftarrow Nodes^{constr}_{start} \cup \bigcup_{bb \in BasicBlocks_{prog,C_i}} (Nodes^{constr}_{bb,in} \cup Nodes^{constr}_{bb,inner} \cup Nodes^{constr}_{bb,out})$

    $DummyNodes \longleftarrow \emptyset$
    for $(nd_1, nd_2) \in Nodes^{constr} \times Nodes^{constr}$ do
        $subpaths \longleftarrow \{\widehat{p} \in \widehat{SubPaths}^{detail,prog,C_i} \mid \widehat{p}(0) \in nd_1 \wedge \widehat{p}(len(\widehat{p})) \in nd_2 \wedge$
                                 $\forall x \in \mathbb{N}_{>0} \cap \mathbb{N}_{<len(\widehat{p})} : \neg\exists nd \in Nodes^{constr} : \widehat{p}(x) \in nd\}$
        $setsOfSubpaths \longleftarrow \{subpaths\}$
        for $(E, BD, cov) \in edgeWeightSens$ do
            $setsOfSubpaths' \longleftarrow \emptyset$
            for $set \in setsOfSubpaths$ do
                for $covMember \in cov$ do
                    $set' \longleftarrow \{\widehat{p} \in set \mid \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wE}^{BD}(\widehat{p}, x) \in covMember\}$
                    if $set' \neq \emptyset$ then
                        $setsOfSubpaths' \longleftarrow setsOfSubpaths' \cup \{set'\}$
            $setsOfSubpaths \longleftarrow setsOfSubpaths'$
        $firstEdge \longleftarrow 1$
        for $set \in setsOfSubpaths$ do
            if $firstEdge = 1$ then
                $targetNd \longleftarrow nd_2$
                $firstEdge \longleftarrow 0$
            else
                $targetNd \longleftarrow$ fresh dummy node
                $DummyNodes \longleftarrow DummyNodes \cup \{targetNd\}$
            $Edges^{constr} \longleftarrow Edges^{constr} \cup \{(nd_1, targetNd)\}$
            for $E \in Events$ do
                $\widehat{wE}^{UB}((nd_1, targetNd)) \longleftarrow \max_{\widehat{p} \in set} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wE}^{UB}(\widehat{p}, x)$
                $\widehat{wE}^{LB}((nd_1, targetNd)) \longleftarrow \min_{\widehat{p} \in set} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wE}^{LB}(\widehat{p}, x)$
            if $targetNd \neq nd_2$ then
                $Edges^{constr} \longleftarrow Edges^{constr} \cup \{(targetNd, nd_2)\}$
                for $E \in Events$ do
                    $\widehat{wE}^{UB}((targetNd, nd_2)) \longleftarrow 0$
                    $\widehat{wE}^{LB}((targetNd, nd_2)) \longleftarrow 0$
    $Nodes^{constr} \longleftarrow Nodes^{constr} \cup DummyNodes$
```

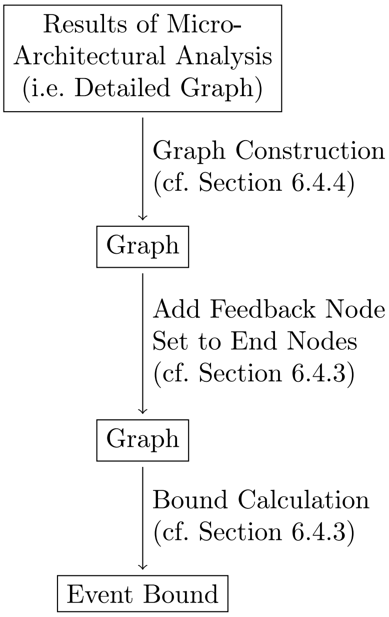


Figure 6.4.: A workflow for the calculation of event bounds.

The exact form of graph G^{constr} depends on the detailed graph $G^{detail,prog,C_i}$ as well as the chosen values along the three aforementioned axes of sensitivity. Thus, from now on, we choose a name for this graph that reflects all the parameters it depends on.

$$G^{prog,C_i,nodeSens_{bndr},nodeSens_{insd},edgeWeightSens} := G^{constr} \tag{6.153}$$

The parametric graph subsumes the detailed graph for all possible choices of sensitivity. A brief proof sketch for this statement is presented on page 297.

$$\begin{aligned}
&\forall nodeSens_{bndr} \in NodeSens_{bndr}^{detail,prog,C_i} : \\
&\quad \forall nodeSens_{insd} \in NodeSens_{insd}^{detail,prog,C_i} : \\
&\qquad \forall edgeWeightSens \in EdgeWeightSens : \\
&\qquad\quad G^{prog,C_i,nodeSens_{bndr},nodeSens_{insd},edgeWeightSens} \models G^{detail,prog,C_i}
\end{aligned} \tag{6.154}$$

In Figure 6.4, we propose a workflow for the calculation of event bounds. It starts from the results of the micro-architectural analysis, which correspond to the detailed graph. As a first step, a graph at the granularity of basic blocks is constructed based on the results of the micro-architectural analysis. The construction follows the principles presented in this subsection. The resulting graph subsumes the detailed graph (cf. equation (6.154)). Thus, according to Section 6.4.3, it is safe to add a feedback node set of the graph to its end nodes and, finally, calculate the event bound based on all paths through the graph. The bound calculation uses lifted system properties in order to prune infeasible paths through the graph.

In the experiments presented in the implementation part of this thesis, we construct graphs at basic-block granularity that are instances of the parametric graph presented in this subsection. Note, however, that our actual implementations of graph construction algorithms differ from the parametric construction proposed by Algorithms 6.1 to 6.3. We have a specialized implementation for the construction of graphs that are insensitive with respect to all three proposed axes of sensitivity. Moreover, we have a specialized implementation that is fully node-sensitive at basic block boundaries, node-insensitive inside of basic blocks, and parametric in the edge-weight-sensitivity. These specialized construction algorithms were implemented before the parametric graph construction had been specified. Thus, it is an open task to implement a graph construction that supports all combinations of sensitivity parameters that the parametric graph supports.

To the best of our knowledge, this subsection contains the first detailed and formally structured discussion on different degrees of detail of graphs to be used in WCET analysis. We pointed out that the degree of detail of a graph (as e.g. determined by the axes of sensitivity that we propose) can have a significant impact on the resulting WCET bound. Overly detailed graphs, however, tend to be prohibitive in terms of runtime and memory consumption needed for the bound calculation (e.g. during implicit path enumeration). Thus, the degree of detail at which graphs are constructed can be used to find a trade-off between analysis precision and analysis efficiency.

6.4.5. ISA-Level Control Flow Properties

In this subsection, we briefly sketch how to safely use control flow properties of the program under analysis in order to detect infeasible abstract traces at the level of approximation of paths through a graph. Intuitively, a control flow property only argues about which basic blocks are executed how often (and potentially also in which order). A loop bound is a typical example for a control flow property.

In Section 6.4.1, we defined the set $\mathit{ContrFlows}_{\mathit{prog},C_i}$ of all possible control flows of a program *prog* when executed on processor core C_i. Each control flow in $\mathit{ContrFlows}_{\mathit{prog},C_i}$ might contain arbitrary instructions of program *prog*. Intuitively, however, a control flow is already uniquely determined by the sequence of basic blocks it corresponds to (cf. equation (6.87)). A sequence of basic blocks, again, can be represented as a corresponding sequence of basic block start instructions. Thus, we can reduce any member of $\mathit{ContrFlows}_{\mathit{prog},C_i}$ to its basic block start instructions without losing any information about the logical control flow.

$$\mathit{ContrFlows}^{\mathit{BB}}_{\mathit{prog},C_i} = \{\mathit{filter}(\mathit{flow}, \lambda \mathit{ins}.\mathit{ins} \in \mathit{BBStarts}_{\mathit{prog},C_i}) \mid \mathit{flow} \in \mathit{ContrFlows}_{\mathit{prog},C_i}\} \tag{6.155}$$

$$\mathit{BBStarts}_{\mathit{prog},C_i} = \{\mathit{front}(\mathit{bb}) \mid \mathit{bb} \in \mathit{BasicBlocks}_{\mathit{prog},C_i}\} \tag{6.156}$$

Note that the sets $\mathit{ContrFlows}_{\mathit{prog},C_i}$ and $\mathit{ContrFlows}^{\mathit{BB}}_{\mathit{prog},C_i}$ are not yet completely independent of the micro-architecture. Consider, e.g., the case that—on a given hardware platform—the execution of a particular instruction *ins* and its successor instruction is always completely overlapped by a preceding instruction. This can happen due to pipelining. In this situation, there is no $\mathit{flow} \in \mathit{ContrFlows}_{\mathit{prog},C_i}$ such that $\mathit{back}(\mathit{flow}) = \mathit{ins}$. A similar pathological scenario exists for $\mathit{ContrFlows}^{\mathit{BB}}_{\mathit{prog},C_i}$ in case the execution of a particular basic block and its successor basic block are always completely overlapped by a preceding basic block.

We can make both sets of control flows independent of the micro-architecture by additionally considering all prefixes of all members they already contain. In the following, we formally define this for set $\mathit{ContrFlows}^{\mathit{BB}}_{\mathit{prog},C_i}$.

$$\mathit{ContrFlows}^{\mathit{BB},\mathit{prefixes}}_{\mathit{prog},C_i} = \bigcup_{\mathit{flow} \in \mathit{ContrFlows}^{\mathit{BB}}_{\mathit{prog},C_i}} \mathit{prefixes}(\mathit{flow}) \tag{6.157}$$

$$\mathit{prefixes} : \mathit{InstrMemAddrList} \to \mathcal{P}(\mathit{InstrMemAddrList}) \tag{6.158}$$

$$\begin{aligned} \mathit{prefixes}(\mathit{list}) = \{&\mathit{list}' \in \mathit{InstrMemAddrList} \\ &\mid \mathit{len}(\mathit{list}') \leq \mathit{len}(\mathit{list}) \land \forall x \in \mathbb{N}_{<\mathit{len}(\mathit{list}')} : \mathit{list}'(x) = \mathit{list}(x)\} \end{aligned} \tag{6.159}$$

The resulting set $\mathit{ContrFlows}^{BB,prefixes}_{prog,C_i}$ contains all control flow prefixes that are possible according to the semantics of the program at the level of the *instruction set architecture* (ISA)—under the assumption that the micro-architecture (i.e. *Cycle*) correctly implements the ISA. It is a subset of all arbitrary sequences of basic block start instructions.

$$\mathit{ContrFlows}^{BB,prefixes}_{prog,C_i} \subseteq \mathit{ArbitrFlows}^{BB,prefixes}_{prog,C_i} \tag{6.160}$$

$$\begin{aligned}\mathit{ArbitrFlows}^{BB,prefixes}_{prog,C_i} = \{&\mathit{flow} \in \mathit{InstrMemAddrList} \\ &\mid \forall x \in \mathbb{N}_{<len(\mathit{flow})} : \mathit{flow}(x) \in \mathit{BBStarts}_{prog,C_i}\}\end{aligned} \tag{6.161}$$

We can specify Boolean predicates P_k on the members of $\mathit{ArbitrFlows}^{BB,prefixes}_{prog,C_i}$. Any such predicate that holds for each member of $\mathit{ContrFlows}^{BB,prefixes}_{prog,C_i}$ is called an *ISA-level control flow property*. Let $\mathit{Prop}^{BB,prefixes}_{prog,C_i}$ be a set of ISA-Level control flow properties.

$$\mathit{Prop}^{BB,prefixes}_{prog,C_i} = \{P_1, \ldots, P_p\} \tag{6.162}$$

$$\forall \mathit{flow} \in \mathit{ContrFlows}^{BB,prefixes}_{prog,C_i} : \forall P_k \in \mathit{Prop}^{BB,prefixes}_{prog,C_i} : P_k(\mathit{flow}) \tag{6.163}$$

Such control flow properties are often specified manually (e.g. by annotating the source code of the program with pragmas [Falk et al., 2016]). It is, however, also possible to automatically determine them by arguing about the semantics of the program at the level of the ISA or a high-level programming language [Ermedahl and Gustafsson, 1997; Healy et al., 1998; Stein and Martin, 2007; Ruiz and Cassé, 2015]. Control flow properties can be specified in different annotation languages [Kirner et al., 2011; Bonenfant et al., 2012], which differ in expressiveness. Control flow properties are typically used during WCET analysis to prune infeasible implicit paths [Li and Malik, 1995; Engblom and Ermedahl, 2000; Raymond, 2014].

In this subsection, we demonstrate how to safely use lifted versions of ISA-level control flow properties to detect infeasible relaxed paths of a graph. As a starting point, we define a control flow pseudo event CF_{ins} per basic block start instruction *ins*. In the same way as the regular system events, each control flow pseudo event shall have a pair of event-bounding edge weights.

$$\mathit{CFEvents}^{BB}_{prog,C_i} = \{\mathit{CF}_{ins} \mid ins \in \mathit{BBStarts}_{prog,C_i}\} \tag{6.164}$$

$$\forall \mathit{CF}_{ins} \in \mathit{CFEvents}^{BB}_{prog,C_i} : \exists \widehat{w\mathit{CF}^{UB}_{ins}}, \widehat{w\mathit{CF}^{LB}_{ins}} \in \widehat{\mathit{Weights}} \tag{6.165}$$

In the detailed graph (cf. Section 6.4.2), the event bounding edge weights $\widehat{w\mathit{CF}^{UB}_{ins}}$ and $\widehat{w\mathit{CF}^{LB}_{ins}}$ shall have the value one for an edge *edg* in case *edg* is an out-edge of an in-node of the basic block of which *ins* is the start instruction. Otherwise, they shall have the value zero.

$$\begin{aligned}&\forall \mathit{CF}_{ins} \in \mathit{CFEvents}^{BB}_{prog,C_i} : \forall BD \in \{UB, LB\} : \forall edg \in \mathit{Edges}^{detail,prog,C_i} : \\ &\widehat{w\mathit{CF}^{BD}_{ins}}(edg) = \Big|\{bb \in \mathit{BasicBlocks}_{prog,C_i} \mid ins = \mathit{front}(bb) \ \wedge \\ &\qquad \exists nd_{in} \in \mathit{Nodes}^{detail,prog,C_i}_{bb,in} : edg \in \mathit{outEdges}(nd_{in})\}\Big|\end{aligned} \tag{6.166}$$

In the subsumption relations (i.e. path subsumption and graph subsumption, cf. Section 6.4.3) and during the construction of graphs (cf. Section 6.4.4), we shall treat the event-bounding edge weights of the control flow pseudo events in the same way as those of the regular system events.

We use the event-bounding edge weights of the control flow pseudo events to map each relaxed path of a graph G^A to the subset of $\mathit{ArbitrFlows}^{BB,prefixes}_{prog,C_i}$ that it describes.

$$\gamma^A_{cf,prog,C_i} : \widehat{\mathit{RelPaths}}^A \to \mathcal{P}(\mathit{ArbitrFlows}^{BB,prefixes}_{prog,C_i}) \tag{6.167}$$

$$\begin{aligned}\gamma^A_{cf,prog,C_i}(\widehat{p^A}) = \{&\mathit{flow} \in \mathit{ArbitrFlows}^{BB,prefixes}_{prog,C_i} \mid \\ &\exists \mathit{part} \in \mathit{Partitionings}(\mathit{len}(\widehat{p^A}), \mathit{len}(\mathit{flow})) : \\ &\quad \forall \mathit{CF}_{ins} \in \mathit{CFEvents}^{BB}_{prog,C_i} : \forall x \in \mathbb{N}_{<\mathit{len}(\widehat{p^A})} : \\ &\quad\quad \widehat{\mathit{wCF}^{LB}_{ins}}(\widehat{p}, x) \\ &\quad\quad \le |\{i \in \mathbb{N} \mid \mathit{from}(\mathit{part}, x) \le i \le \mathit{to}(\mathit{part}, x) \wedge \mathit{ins} = \mathit{flow}(i)\}| \\ &\quad\quad \le \widehat{\mathit{wCF}^{UB}_{ins}}(\widehat{p}, x)\}\end{aligned} \tag{6.168}$$

Subsequently, we slightly extend the definition of relation *PathDescrTrace* (cf. equation (5.69)) with a conjunction in a way that a subpath can only describe those sequences of abstract states whose control flow reduced to basic block start instructions is an extension of one of the flows described by the path according to $\gamma^A_{cf,prog,C_i}$. Note that the soundness statements made in the previous subsections still hold after this extension.

It follows that a path which is infeasible with respect to $\mathit{ContrFlows}^{BB,prefixes}_{prog,C_i}$ is also infeasible with respect to $\mathit{ExecRuns}_{prog,C_i}$.

$$\begin{aligned}&\gamma^A_{cf,prog,C_i}(\widehat{p^A}) \cap \mathit{ContrFlows}^{BB,prefixes}_{prog,C_i} = \emptyset \\ \Rightarrow &[\bigcup_{\widehat{t} \in \gamma^A_{path,prog,C_i}(\widehat{p^A}) \cap \widehat{\mathit{Traces}_{prog,C_i}}} \gamma_{trace,prog,C_i}(\widehat{t})] \cap \mathit{ExecRuns}_{prog,C_i} = \emptyset\end{aligned} \tag{6.169}$$

As a consequence, paths which are infeasible with respect to $\mathit{ContrFlows}^{BB,prefixes}_{prog,C_i}$ can safely be pruned during the calculation of event bounds. In order to detect such infeasible paths, we lift control flow properties $P_k \in \mathit{Prop}^{BB,prefixes}_{prog,C_i}$ to the level of approximation of paths through a graph. The lifted versions shall respect the following soundness criterion.

$$\forall \widehat{p^A} \in \widehat{\mathit{SubPaths}}^A : [\, \exists \mathit{flow} \in \gamma^A_{cf,prog,C_i}(\widehat{p^A}) : P_k(\mathit{flow}) \,] \Rightarrow \widehat{P^{path}_k}(\widehat{p^A}) \tag{6.C11}$$

The lifted versions of ISA-level control flow properties are used for the detection of infeasible paths in the same way as the lifted version of regular system properties. Moreover, they are also lifted to the level of approximation of implicit path enumeration in the same way by applying the criterion respectively the lifting rule presented in Section 5.4.

This concludes the technical contribution of this subsection. We continue by discussing the related work. Vincent Mussot and Pascal Sotin propose to represent a control flow property as an automaton which generates the language of possible control flows that fulfill the property [Mussot and Sotin, 2015]. They obtain the graph on which the event bound is calculated by performing an automata product between the CFG and the automata representing the different control flow properties. Their approach is an instance of our framework as the automata product corresponds to the application of lifted versions of the control flow properties. It is, however, unclear how to adapt the approach to graphs that do not coincide with the CFG (e.g. for the sake of precision, cf. Section 6.4.4).

Criterion (6.C11), in contrast, does not make any assumptions about the degree of detail of the considered graph. In particular, it is applicable to graphs that are not at the granularity of basic blocks (i.e. an edge can argue about more than one basic block execution). Such graphs can e.g. be the result of graph transformations that aim at compressing a graph in order to reduce

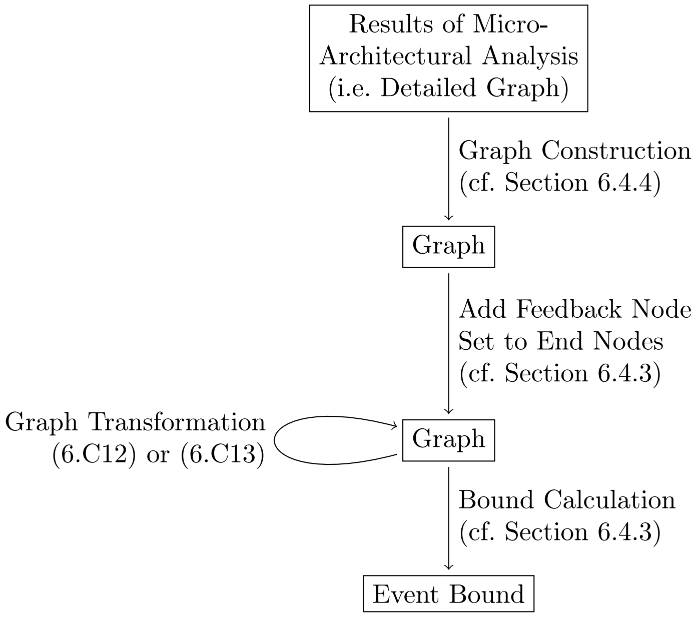


Figure 6.5.: An extended workflow for the calculation of event bounds. It additionally incorporates graph transformations before the actual bound calculation.

the runtime and/or memory consumption of a subsequent implicit path enumeration. Ingmar Stein presents such graph compression techniques in his dissertation [Stein, 2010]. He, however, does not describe how to safely apply control flow properties to detect infeasible paths through the compressed graphs. We close this gap with the presented formal criterion.

Be aware that the technical contribution of this subsection is only a brief sketch (due to time and space restrictions). A more detailed formal presentation and a full soundness proof exceed the scope of this thesis.

6.4.6. Toward Graph Transformations

> Der Graf ist nicht das, was er mal war.
> Ja, der Graf wirkt heut' seltsam und bizarr.
>
> *(Der Graf, Die Ärzte, 1998)*

In Section 6.4.3, we propose to determine a set of feedback nodes for a graph G^B and to add it to its set of end nodes. It is safe to calculate an event bound based on all paths through the resulting graph $G^{B\prime}$ (i.e. based on $\widehat{Paths}^{B\prime}$) in case G^B subsumes the detailed graph. Figure 6.4 sketches this workflow.

In this subsection, we extend this workflow by additionally incorporating graph transformations after the set of feedback nodes has been added to the end nodes. Graph transformations are e.g. used to compress a graph and, thus, make a subsequent implicit path enumeration more efficient. A correspondingly extended sketch of the workflow is depicted in Figure 6.5.

We can define a simple *soundness criterion* for graph transformations based on the notion of *path subsumption* (cf. equation (6.120)). Intuitively, a graph transformation is sound if every path through the original graph G^C is subsumed by a path through the transformed graph $G^{C\prime}$.

$$\forall \widehat{p} \in \widehat{Paths^C} : \exists \widehat{p'} \in \widehat{Paths^{C\prime}} : \widehat{p'} \vDash_{path} \widehat{p} \tag{6.C12}$$

According to statement (6.121), any path which subsumes a feasible path has to be feasible itself. It follows from the proofs of statements (6.129), (6.130), and (6.137) that the event bounds calculated based on the transformed graph are also sound with respect to the concrete traces. We omit the formal details for the sake of readability.

Note that the event bounds calculated based on a graph resulting from a graph transformation guaranteeing criterion (6.C12) might—in some cases—be strictly more precise than those calculated based on the graph before the transformation. For most practical cases, however, we can expect the lifted properties to be monotone with respect to path subsumption, i.e.:

$$[\widehat{p'} \vDash_{path} \widehat{p}] \Rightarrow [\widehat{P_k^{path}}(\widehat{p}) \Rightarrow \widehat{P_k^{path}}(\widehat{p'})] \tag{6.170}$$

In case this monotonicity is given, a graph transformation guaranteeing criterion (6.C12) cannot improve the precision of the resulting event bounds.

Another simple *soundness criterion* for graph transformations is based on the *pruning of provably infeasible nodes and edges*. A node respectively an edge of a graph is considered infeasible if and only if every path through the graph containing it is infeasible. Intuitively, a graph transformation is sound if it creates a subgraph $G^{C\prime}$ of the original graph G^C such that every path through G^C not detected as infeasible by the lifted properties is also a path through $G^{C\prime}$.

$$G^{C\prime} \subseteq_{pairwise} G^C \wedge \widehat{Paths^{C\prime}} \supseteq \{\widehat{p} \in \widehat{Paths^C} \mid \forall P_k \in Prop_{prog,C_i} : \widehat{P_k^{path}}(\widehat{p})\} \tag{6.C13}$$

A transformation guaranteeing criterion (6.C13) leaves the event bounds (calculated directly on $\widehat{Paths^{C\prime}}$) unchanged (compared to calculating them directly on $\widehat{Paths^C}$) as it only prunes nodes and edges that only belong to paths that would be pruned anyway during the bound calculation (cf. equations (6.127) and (6.128)).

The dissertation of Ingmar Stein [Stein, 2010] presents graph transformations that *compress a graph* with the goal of reducing the runtime and memory consumption of a subsequent implicit path enumeration. Intuitively, these transformations fulfill the criteria that we present. His formalism, however, only argues about a single type of edge weight per graph—which is maximized respectively minimized to calculate an event bound. The safe incorporation of system properties for the detection of infeasible paths through the graph is only vaguely discussed. Stein only briefly mentions that parts of the graph on which cache persistence constraints argue shall not be compressed. It is e.g. not clear how to safely incorporate control flow properties (typically arguing about which basic blocks are executed) in case the edges in a graph are compressed across basic block boundaries (i.e. the compressed graph is no longer at basic block granularity). Thus, we believe that the soundness of Stein's approach relies on implicit assumptions that are not documented in his dissertation. We close this gap by the principle of property lifting and the presented formal soundness criteria for graph transformations.

Note that graph transformation can also be applied to improve the precision of event bounds obtained by a subsequent implicit path enumeration [Mussot et al., 2016]. The key idea is to unfold a graph (which trivially fulfills criterion (6.C12)) in order to be able to express system properties in implicit path enumeration that could not have been expressed (or only very imprecisely) in an implicit path enumeration on the original graph. Intuitively, this makes some subpaths of the graph more explicit and, thus, the subsequent implicit path enumeration de facto a bit less implicit. For a practical example based on control flow properties, we refer to a paper by Pascal Raymond [Raymond, 2014]. Mussot et al. exploit this principle by calculating the automata

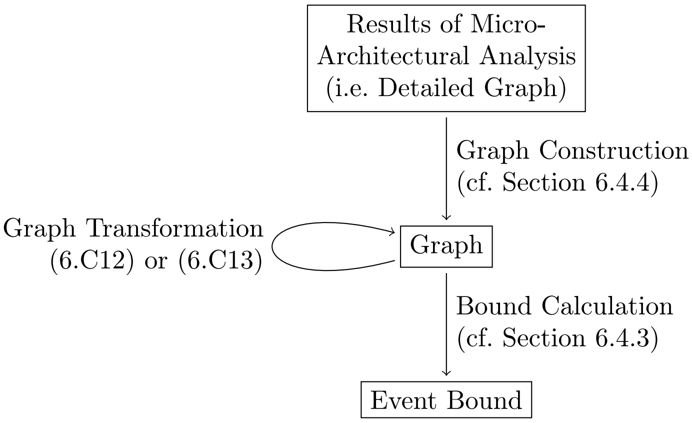


Figure 6.6.: Simplified workflow for the calculation of event bounds. It no longer adds a set of feedback nodes to the end nodes of the graph. It can safely be applied in case the program under analysis is guaranteed to terminate.

product between the CFG and automata representing different control flow properties [Mussot and Sotin, 2015]. In this way, they unfold (criterion (6.C12)) the CFG and prune (criterion (6.C13)) parts of the resulting graph which are provably not included in a feasible path.

6.4.7. Simplification under Guaranteed Termination

According to the soundness proof (on page 289) for the event bounds presented in Section 6.4.3, we only incorporate a set of feedback nodes during the event bound calculation to also be sound with respect to the concrete traces which are not a prefix of a terminated program execution run. If the program under analysis, however, is guaranteed to terminate, there are no concrete traces which are not a prefix of a terminated run.

$$\max_{t \in \mathit{ExecRuns}_{prog,C_i}} \mathit{len}(t) \in \mathbb{N} \Rightarrow \mathit{ExecRuns}^{diverg}_{prog,C_i} = \emptyset \tag{6.171}$$

As a consequence, the event bound calculation for programs that are guaranteed to terminate only has to take into account all paths through the graph (i.e. from a start node to an end node). Adding a set of feedback nodes to the end nodes of the graph is not needed in this case. The correspondingly simplified workflow for the calculation of event bounds for programs that are guaranteed to terminate is depicted in Figure 6.6. Note that this simplified workflow coincides with the classical approach of only arguing about all paths from the start to the end of a program (cf. e.g. [Li and Malik, 1995]).

6.5. Event Bounds Based on Implicit Path Enumeration

6.5.1. The General Case

In the following, we assume that G^B is a graph which can safely be used for the calculation of bounds for event E according to our workflow (cf. Figure 6.5, Figure 6.6). For a bound calculation based on implicit path enumeration, we only consider the implicit paths through G^B for which all lifted properties hold.

$$\widehat{\mathit{LessImplicit}}^B = \{\widehat{i} \in \widehat{\mathit{Implicit}}^B \mid \forall P_k \in \mathit{Prop}_{prog,C_i} \cup \mathit{Prop}^{BB,prefixes}_{prog,C_i} : \widehat{P_k^{impli}}(\widehat{i})\} \tag{6.172}$$

Based on $\widehat{LessImplicit}^B$, we specify upper and lower bounds for event E. Let tt denote the *timesTaken*-component of an implicit path in the following.

$$\widehat{Maximum}^{B,impli}_{prog,C_i,E} = \max_{(tt,*,*)\in\widehat{LessImplicit}^B} \sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(edg) \tag{6.173}$$

$$\widehat{Minimum}^{B,impli}_{prog,C_i,E} = \min_{(tt,*,*)\in\widehat{LessImplicit}^B} \sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(edg) \tag{6.174}$$

The soundness of these event bounds is a consequence of the soundness of the corresponding bounds calculated on paths through the graph, the soundness of implicit path enumeration, and the soundness of property lifting. Thus, the actual bound calculation in our proposed workflow (cf. Figure 6.5, Figure 6.6) can safely be replaced by a calculation based on implicit path enumeration.

Implicit path enumeration inherently approximates away the order in which the edges of a path occur (cf. equation (5.81)). As a consequence, properties arguing about the order of events cannot be expressed in implicit path enumeration.

Implicit path enumeration is typically encoded as *integer linear program* [Li and Malik, 1995]. Such an encoding only supports sets of lifted properties that can be represented as a conjunction of linear constraints. Thus, an ILP encoding further reduces the expressiveness of implicit path enumeration. Pascal Raymond aims at exploring these inherent limits [Raymond, 2014]. He explains how to express control flow properties as precisely as possible for ILP-based implicit path enumeration.

There are also approaches which circumvent these inherent limits of expressiveness at the cost of an increased runtime and/or memory consumption. The first paper about implicit path enumeration [Li and Malik, 1995] already proposes to convert a property set containing disjunctions to a disjunctive normal form. Subsequently, each disjoint property set is used in a separate implicit path enumeration and the actual event bound is obtained by taking the maximum (respectively minimum) over the separate event bounds. It has also been proposed [Mussot and Sotin, 2015] to use graph transformations (cf. Section 6.4.6) to hard-wire certain properties into the structure of the graph. Finally, modern ILP solvers[2] also have native support for quadratic constraints and further logical operators (like disjunction and implication).

6.5.2. The Special Case of WCET Bounds

The calculation of WCET bounds at the level of approximation of paths through a graph by only considering paths from the program start to the program end is not sound for programs that can diverge. A corresponding counter example is given at the end of Section 3.3. If we cannot guarantee the termination of the considered program, however, the calculation of a WCET bound based on all paths through a graph is provably sound in case we have added a feedback node set of the graph to its set of end nodes (cf. Figure 6.5).

Surprisingly, at the level of approximation of implicit path enumeration, a corresponding unsoundness of only considering paths from the program start to the program end does not exist in case the implicit path enumeration solely relies on linear constraints (i.e. no additional features of modern solvers like SOS constraints or indicator constraints are used). Intuitively, as soon as there is an unbounded loop in the graph representation, the objective of the implicit path enumeration calculating the WCET bound is guaranteed to be unbounded in case the implicit path enumeration solely relies on linear constraints.

As a consequence, for the calculation of a WCET bound via implicit path enumeration, it is not necessary to add a feedback node set of the graph to its set of end nodes in case the implicit path enumeration solely relies on linear constraints. Thus, if the implicit path enumeration is

[2] http://www.gurobi.com, https://www.ibm.com/software/commerce/optimization/cplex-optimizer

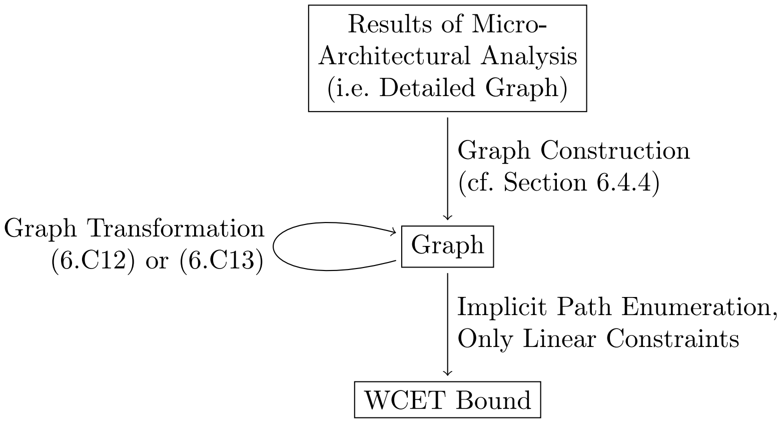

Results of Micro-Architectural Analysis (i.e. Detailed Graph)
Graph Construction (cf. Section 6.4.4)
Graph Transformation (6.C12) or (6.C13)
Graph
Implicit Path Enumeration, Only Linear Constraints
WCET Bound

Figure 6.7.: Simplified workflow for the calculation of WCET bounds. It no longer adds a set of feedback nodes to the end nodes of the graph. It can safely be applied in case the WCET bound is calculated via implicit path enumeration solely relying on linear constraints.

guaranteed to only use linear constraints, we can use the simplified WCET bound calculation workflow depicted in Figure 6.7. Note that this simplified workflow coincides with the classical approach of only arguing about all paths from the start to the end of a program (cf. e.g. [Li and Malik, 1995]).

A practical example of how implicit path enumeration effectively *repairs* the potential unsoundness is presented at the end of Section 3.3.

In this section, we would like to sketch a proof construction demonstrating that an implicit path enumeration solely relying on linear constraints is guaranteed to never calculate a finite WCET bound for a program that can diverge. For our proof construction, we assume that there is a program that can diverge. Now, we choose one of its concrete traces that can be extended indefinitely without ever reaching the program end. As we assume a finite set of possible concrete system states, there must be an extension of the chosen concrete trace in which one of the concrete system states occurs at least twice. Thus, we have a loop formed of concrete system states and the transitions between these states. Due to the soundness of the micro-architectural analysis and the graph construction, there must be a loop in the constructed graph which describes the concrete loop. Moreover, due to the soundness of property lifting, there cannot be an absolute or relative loop bound for the loop in the graph representation because this would mean that there were no feasible paths from the start node of the graph representation that take the loop indefinitely often. However, there have to be such feasible paths as they describe the concrete traces taking the concrete loop indefinitely often. As an intermediate result, we have a graph representation with a loop that has no absolute or relative loop bound. Thus, if we have no absolute or relative loop bound, the only way to avoid that the loop of the graph representation holds itself in an implicit path (without having to be connected to the remainder of the implicit path) would be a constraint of the following form.

$$timesTaken_{edge_{outer}} = 0 \Rightarrow timesTaken_{edge_{loop}} = 0$$

To the best of our knowledge, such an implication constraint can only be implemented solely based on linear constraints if the variable on the right-hand side of the implication has a limited value range (Big-M approach). As there cannot be a loop bound, however, no edge in the loop of the graph representation can have such a limited value range. As a result: By solely relying on linear constraints, we cannot avoid that the unbounded loop in the graph representation holds itself (and, thus, can be executed indefinitely often) during implicit path enumeration. Hence, the maximized objective is necessarily unbounded.

This short proof sketch shows that the soundness of implicit path enumeration for the calculation of WCET bounds for programs that are not guaranteed to terminate is mostly due to a limited expressiveness of the linear constraints. We are able to show that the incorporation of implication-like constraints (if no in-edge of the loop is taken, no edge inside of the loop must be taken, implemented via SOS constraints or indicator constraints) can lead to a finite WCET bound for a program that can diverge if no feedback node set was added to the end node of the graph before the implicit path enumeration: For the example at the end of Section 3.3, such a (soundly lifted) implication-like constraint could look as follows.

$$\mathit{timesTaken}_{BB_3} = 0 \Rightarrow \mathit{timesTaken}_{BB_4} = 0$$

We used Gurobi to verify that this actually leads to a finite WCET bound for the example program which diverges. Thus, relying on all the types of constraints that modern ILP solvers have to offer, a WCET bound calculated via implicit path enumeration is not guaranteed to be sound if only paths from the start to the end of the program are considered. Solely relying on linear constraints during implicit path enumeration, however, seems to be sound (as sketched above). In case it should be necessary to rely on advanced solver features for an improved precision or performance, we recommend resorting to the feedback node set approach instead.

Chapter 7 Multi-Core Processors with Shared Buses

> But there's a huge interference,
> They're saying "You shouldn't hear it."
>
> *(Renegade, Jay-Z & Eminem, 2001)*

Multi-core processors share common resources (like e.g. caches and buses) between multiple processor cores. In this way, they provide a trade-off between the computation capacity of multi-processor systems and the low cost, weight, and energy consumption of single-processor systems.

From a timing-verification point of view, however, multi-core processors are significantly more challenging than single-core processors: the execution time of a program executed on a particular processor core depends on the programs simultaneously executed on the concurrent cores [Abel et al., 2013]. This phenomenon is typically referred to as *shared-resource interference.* For a detailed discussion of the existing work in the area of timing verification for multi-core processors, we refer to Chapter 2.

In this chapter, we propose to model shared-bus interference by non-determinism and, thus, to safely account for the effect that a unit of interference can have on the pipeline of the interfered processor core. This way of modeling is processor-core-modular and supports hardware platforms exhibiting timing anomalies. Based on it, we demonstrate the calculation of *co-runner-insensitive* WCET bounds—following the Murphy approach—and *co-runner-sensitive* WCET bounds—relying on a cumulative approximation of the shared-resource access behavior of the concurrent cores.

For this thesis, we assume the following schematic system design for multi-core processors with shared buses.

7.1. Schematic System Design

We consider a multi-core processor with at least two processor cores. Let *Cores* denote its set of processor cores.

$$\mathit{Cores} = \{C_1, C_2, \ldots\} \tag{7.1}$$

$$|\mathit{Cores}| \geq 2 \tag{7.2}$$

The multi-core processor shall have one shared bus that connects all cores to the memory. This schematic design is sketched in Figure 7.1.

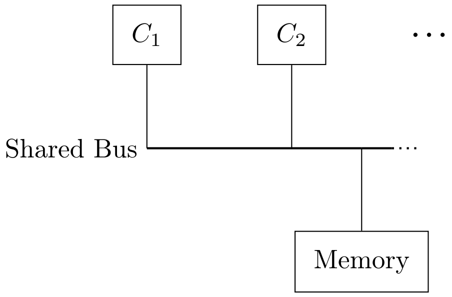


Figure 7.1.: Schematic system design of a multi-core processor with a shared bus.

Note that this design is relatively general as it does not make any assumptions about the shared memory. The memory could be a simple SRAM that takes a constant amount of clock cycles per data access—independently of the access history. However, the memory could as well consist of a cache and a DRAM—which can lead to further interference between the processor cores.

Further note that more complex system designs with respect to the interconnect (e.g. featuring a hierarchical combination of multiple buses or a network-on-chip) could be modeled with techniques similar to those presented in this thesis. For simplicity, however, we assume that a single bus is the only bandwidth resource leading to interference between the cores.

In this thesis, we focus on safely modeling shared-bus interference. As a consequence, the experiments presented in the implementation part of this thesis assume a simple SRAM as shared memory. Note, however, that our proposed way of modeling the shared-bus interference can also be reused in analyses that additionally account for other sources of interference (cf. Section 7.8).

For the rest of this thesis, we rely on the following *notational conventions* with respect to the processor cores.

- C_i refers to the processor core for which we calculate a WCET bound
- C_j refers to one of the concurrent processor cores of C_i, i.e. $C_j \in \mathit{Cores} \setminus \{C_i\}$
- C_k is used if we argue about all processor cores (including C_i)

Finally, for the sake of simplicity, we assume that there is no preemptive scheduling used on the processor cores of the system under analysis. In this way, we can safely omit some formal machinery (in particular the shorthand events $\mathit{Event}_{prog,C_i,E}$, cf. equation (6.19)) in order to focus on the essential aspects with respect to the consideration of the shared-resource interference.

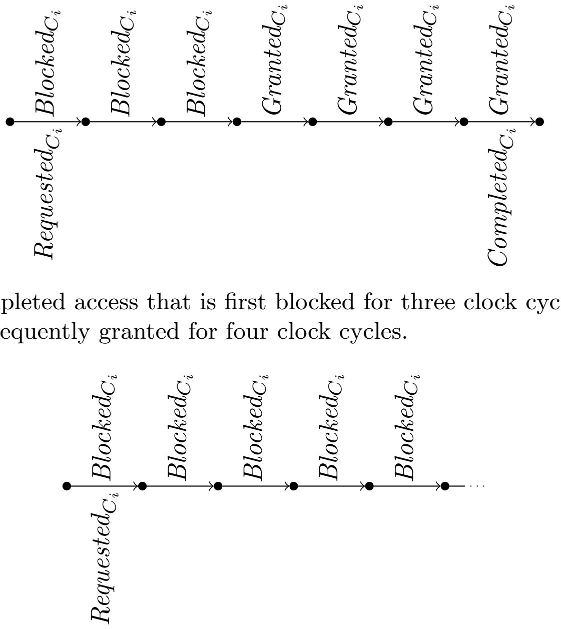
7.2. A Baseline Abstract Model

In order to safely model the bus access behavior of processor core C_i, first, we take a look at what a bus access of core C_i on the concrete system means in terms of system events. Every bus access of core C_i begins with a corresponding *access request* of C_i. For every bus access, there is only a single occurrence of the corresponding system event $\mathit{Requested}_{C_i}$ at the first cycle of the access.

$$\mathit{Requested}_{C_i} \in \mathit{Events} \tag{7.3}$$

The event $\mathit{Requested}_{C_i}$ means that—starting from (including) its occurrence—the bus arbiter is aware of the request by core C_i. The arbiter may choose to immediately *grant access* to core C_i, or to *block access* of core C_i for an arbitrary (potentially infinite) amount of clock cycles.

$$\mathit{Blocked}_{C_i}, \mathit{Granted}_{C_i} \in \mathit{Events} \tag{7.4}$$



(a) Completed access that is first blocked for three clock cycles and subsequently granted for four clock cycles.

(b) Access that is blocked indefinitely.

Figure 7.2.: Two examples of bus accesses of processor core C_i on a concrete system.

Once an access has been granted, however, it cannot be blocked again. Intuitively, a granted bus access cannot be interrupted or canceled in favor of an access of a different processor core.

Core C_i might *complete an access* that has been granted for at least one clock cycle. There can be at most one access completion per access of core C_i. A completion marks the end of an access.

$$\mathit{Completed}_{C_i} \in \mathit{Events} \tag{7.5}$$

Figure 7.2a shows a completed access that is first blocked for three clock cycles and subsequently granted for four clock cycles. Note, however, that an access might never complete. It could as well be blocked indefinitely (cf. Figure 7.2b, e.g. under non-fair bus arbitration) or granted indefinitely (e.g. because behind the considered bus there is another bus with non-fair arbitration).

We say that processor core C_i has a *pending bus access* at a certain instant if it is either blocked at the bus or granted access to it.

$$\mathit{Pending}_{C_i} = \mathit{Granted}_{C_i} \dot{\cup} \mathit{Blocked}_{C_i} \tag{7.6}$$

Note that corresponding bus access events also exist per concurrent processor core C_j. For the sake of simplicity, however, these events are not explicitly specified.

In general, it depends on the programs on the concurrent cores for how long an access is blocked, for how long it is granted, and whether it completes. As discussed in Chapter 2, however, a fully integrated analysis of the programs on all processor cores does not scale to real-world systems. Thus, we specify a processor-core-modular baseline analysis which models the bus access behavior in a very pessimistic way. It will serve as starting point for the derivation of more precise analyses.

Our baseline abstract model is an extension of the current state of the art in micro-architectural modeling for single-core processors [Thesing, 2004]. In the same way as Thesing, we model the content of the different pipeline stages exactly and use a less precise modeling scheme for the local caches of core C_i. Moreover, however, we model the bus access behavior of core C_i by distinguishing all sequences of bus access events (cf. Figure 7.2) that may arise for core

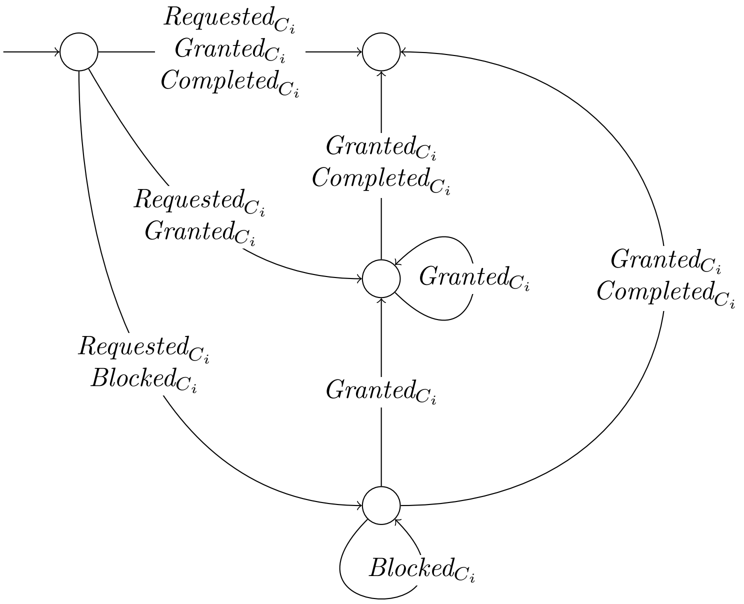


Figure 7.3.: Diagram representation of the non-determinism that we use for pessimistically modeling the bus access behavior of core C_i.

C_i—independently of the programs on the concurrent cores or the details about the resource sharing. This leads to a significant increase of the *non-determinism* during micro-architectural analysis.

The diagram in Figure 7.3 represents this additional non-determinism. Intuitively, a pending bus access that has not yet been granted can be blocked for one more cycle, or granted. If a pending bus access is granted an access cycle, this access cycle can either complete the access, or require at least one more granted access cycle until completion. Note that the micro-architectural analysis also tracks the impact that the respective non-deterministic decisions have on the state of the processor core (i.e. on the contents of the processor core's pipeline stages).

The resulting baseline abstract model is referred to as $\widehat{ExecRuns}^{C_i}_{prog,C_i}$. It models the concrete traces of program *prog* when executed on processor core C_i by only considering the operation of core C_i. It is sound as it covers all possible cases—independently of many details about the concrete system as e.g. the number of concurrent processor cores, their micro-architectural details, the programs they execute, or further details about the resource sharing. Thus, $\widehat{ExecRuns}^{C_i}_{prog,C_i}$ is *processor-core-modular* as it does not make any assumptions about the concurrent processor cores. Moreover, it is sound with respect to a whole family of concrete systems, whose members e.g. only differ in the overall number of processor cores, the bus arbitration policy, or the size of the shared cache.

However, $\widehat{ExecRuns}^{C_i}_{prog,C_i}$ is so far useless for WCET analysis: any bus access could be delayed indefinitely without ever being completed (cf. Figure 7.3). As a consequence, we have to combine $\widehat{ExecRuns}^{C_i}_{prog,C_i}$ with lifted versions of system properties that effectively upper-bound the bus access delay and, thus, enable the determination of finite WCET bounds. We demonstrate this in Section 7.4 and Section 7.5.

Note that, during micro-architectural analysis, we model the bus access events of processor core C_i exactly. As a consequence, the corresponding sets of may-events and must-events coincide for each of these events. Intuitively, we know precisely for each transition between abstract states which of the bus access events occurred, and which did not.

$$\forall E_{bus} \in \{\mathit{Requested}_{C_i}, \mathit{Blocked}_{C_i}, \mathit{Granted}_{C_i}, \mathit{Completed}_{C_i}\} : \quad \widehat{E_{bus}^{UB}} = \widehat{E_{bus}^{LB}} \tag{7.7}$$

Moreover, we provide helper notation to argue about the different accesses of a concrete trace in a convenient way. To this end, we identify each bus access in a concrete trace with the position at which the corresponding access request occurs. Note that this is sufficient since our software convention which prevents multiple program runs from overlapping (cf. end of Section 6.1) guarantees that a program run on core C_i cannot begin while an earlier requested access of core C_i is still pending.

$$\begin{aligned} &\forall t \in \mathit{ExecRuns}_{prog,C_i} : \\ &\quad \mathit{getAccesses}_{C_i}(t) = \{x \in \mathbb{N}_{<len(t)} \mid \mathit{Requested}_{C_i}(t, x)\} \end{aligned} \tag{7.8}$$

Each bus access is assigned the clock cycles during which it is pending.

$$\begin{aligned} &\forall t \in \mathit{ExecRuns}_{prog,C_i} : \forall acc \in \mathit{getAccesses}_{C_i}(t) \\ &\quad \mathit{getPending}_{C_i}(t, acc) = \{x \in \mathbb{N}_{\geq acc} \cap \mathbb{N}_{<len(t)} \mid \mathit{Pending}_{C_i}(t, x) \wedge \\ &\qquad\qquad\qquad\qquad \neg\exists y \in \mathbb{N}_{>acc} \cap \mathbb{N}_{\leq x} : \mathit{Requested}_{C_i}(t, y)\} \end{aligned} \tag{7.9}$$

Similarly, each bus access is assigned the clock cycles during which it is blocked, granted, respectively at which it completes.

$$\begin{aligned} &\forall t \in \mathit{ExecRuns}_{prog,C_i} : \forall acc \in \mathit{getAccesses}_{C_i}(t) \\ &\quad \mathit{getBlocked}_{C_i}(t, acc) = \{x \in \mathit{getPending}_{C_i}(t, acc) \mid \mathit{Blocked}_{C_i}(t, x)\} \wedge \\ &\quad \mathit{getGranted}_{C_i}(t, acc) = \{x \in \mathit{getPending}_{C_i}(t, acc) \mid \mathit{Granted}_{C_i}(t, x)\} \wedge \\ &\quad \mathit{getCompleted}_{C_i}(t, acc) = \{x \in \mathit{getPending}_{C_i}(t, acc) \mid \mathit{Completed}_{C_i}(t, x)\} \end{aligned} \tag{7.10}$$

Since we model the bus access events of the considered processor core exactly in our baseline abstract model (cf. equation (7.7)), the same helper notation can also be specified exactly for sequences of abstract states. Thus, in the following, we use this helper notation on sequences of abstract states as well.

The experiments presented in the implementation part of this thesis assume a simple SRAM as shared memory (cf. Section 7.1). As a consequence, there is a fixed latency LAT which every granted bus access takes until completion. The exact value of this latency is typically determined by the characteristics of the SRAM hardware component.

$$\begin{aligned} &\exists LAT \in \mathbb{N}_{\geq 1} : \\ &\quad \forall t \in \mathit{ExecRuns}_{prog,C_i} : \forall acc \in \mathit{getAccesses}_{C_i}(t) \\ &\qquad \left|\mathit{getGranted}_{C_i}(t, acc)\right| \leq LAT \wedge \\ &\qquad \left|\mathit{getGranted}_{C_i}(t, acc)\right| < LAT \Rightarrow \mathit{getCompleted}_{C_i}(t, acc) = \emptyset \end{aligned} \tag{7.11}$$

A fixed latency LAT can be exploited to reduce the degree of non-determinism in modeling the bus access behavior. The diagram in Figure 7.4 is a correspondingly simplified version of the diagram in Figure 7.3—assuming $LAT = 2$.

Analogous helper functions can also be specified for the access behavior of the concurrent cores C_j. Note, however, that we still consider execution runs of program *prog* on core C_i (as this is what we calculate a WCET bound for). This means that a program run on core C_i can begin while an earlier requested access of core C_j is still pending. Consequently, such an access cannot

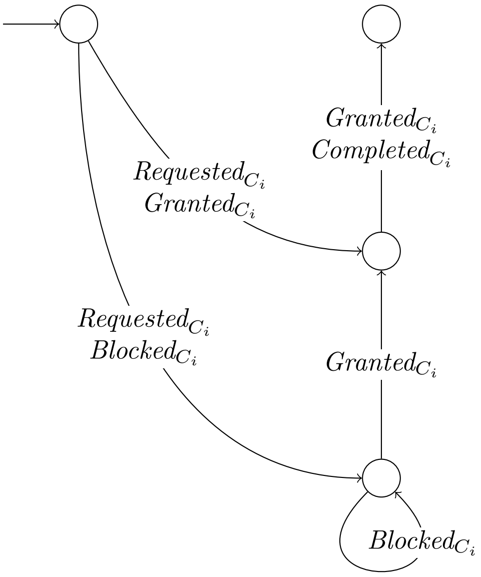


Figure 7.4.: The degree of non-determinism for modeling the bus access behavior can be reduced if every granted bus access takes a fixed latency until completion. In this figure, this is demonstrated for a fixed latency of two clock cycles.

be identified by the position of its request since this position would be before the start of the program run. Instead, such an access is identified by the front position of the program run.

$$\begin{aligned} &\forall t \in \mathit{ExecRuns}_{prog,C_i} : \forall C_j \in \mathit{Cores} \setminus \{C_i\} : \\ &\quad \mathit{getAccesses}_{C_j}(t) = \{x \in \mathbb{N}_{<len(t)} \mid \mathit{Requested}_{C_j}(t,x)\} \cup \\ &\qquad\qquad \{x \in \mathbb{N}_{\leq 0} \cap \mathbb{N}_{<len(t)} \mid \mathit{Pending}_{C_j}(t,x)\} \end{aligned} \tag{7.12}$$

7.3. Running Example: Round-Robin Bus Arbitration

In case multiple processor cores have pending bus accesses that have not yet been granted, a *bus arbitration policy* has to determine which of them is granted access to the bus first. If the decisions made by an arbitration policy depend on the bus access history, the arbitration policy is called *event-driven*. *Time-division multiple access* (TDMA) bus arbitration is not event-driven as the bus access history has no impact on its arbitration decisions. For a more detailed classification of the existing bus arbitration policies, we refer to a survey on shared-resource interference [Abel et al., 2013]. In this thesis, we solely consider event-driven bus arbitration protocols. Nonetheless, the principles presented in this thesis could as well be used to derive analyses for systems with TDMA bus arbitration or hybrid variants (i.e. mixing event-driven and TDMA schemes).

Round-Robin is a popular event-driven bus arbitration policy (cf. Appendix F of [Hennessy and Patterson, 2011]). The Round-Robin policy relies on a fixed total order among the processor cores. In case bus access has not been granted to any processor core during the preceding clock cycle, the arbitration decision for the current clock cycle selects the first processor core with a pending bus access according to the total order. In case bus access has been granted to processor core C_k during the preceding clock cycle, the arbitration decision for the current clock cycle selects the next processor core with a pending bus access following C_k according to the total order. If there are no cores with a pending bus access following C_k in the total order, the arbitration decision selects the first processor core with a pending bus access according to the total order. For a detailed formal specification of the Round-Robin arbitration policy, we refer to a textbook [Kovalev et al., 2014].

Round-Robin bus arbitration will serve as a running example during the remainder of this thesis. Now, we present two key properties of it, which will be used during the derivation of our WCET analyses (cf. Section 7.4 and Section 7.5). Note that existing approaches to timing verification for multi-core processors with a Round-Robin bus arbitration (e.g. [Pellizzoni et al., 2010; Altmeyer et al., 2015]) are also implicitly based on these properties.

For Round-Robin bus arbitration, there is a *local upper bound* on the number of blocked cycles per bus access. Intuitively, the worst thing that can happen for a processor core is that each concurrent core is granted a full access first. This is expressed by the following property P_{rr}.

$$\begin{aligned} P_{rr}(t) \Leftrightarrow & \forall C_k \in \mathit{Cores} : \forall \mathit{acc} \in \mathit{getAccesses}_{C_k}(t) : \\ & \left|\mathit{getBlocked}_{C_k}(t, \mathit{acc})\right| \leq (|\mathit{Cores}| - 1) \cdot \mathit{LAT} \end{aligned} \tag{7.13}$$

Moreover, Round-Robin bus arbitration is a *work-conserving* arbitration policy. This means that a bus access of a processor core can only be blocked at a particular instant if another core is granted access at the same instant. This is expressed by the following property P_{wc}.

$$\begin{aligned} P_{wc}(t) \Leftrightarrow & [\forall C_k \in \mathit{Cores} : \forall x \in \mathbb{N}_{<\mathit{len}(t)} : \\ & \mathit{Blocked}_{C_k}(t, x) \Rightarrow \exists C_j \in \mathit{Cores} \setminus \{C_k\} : \mathit{Granted}_{C_j}(t, x)] \end{aligned} \tag{7.14}$$

Note that all properties of Round-Robin bus arbitration that this thesis relies on also hold for the *Least-Recently-Granted* policy, which is *age-based* (cf. Appendix F of [Hennessy and Patterson, 2011]). Thus, we can safely reuse the analyses relying on these properties for systems with Least-Recently-Granted bus arbitration. The Least-Recently-Granted policy grants access to the core that has waited the longest amount of time since its last granted access [Satpathy et al., 2012]. If there are multiple cores that have not yet been granted access to the bus since system start, an arbitration decision among them shall be implementation-defined (e.g. following a fixed total order among the cores).

Further note that the term Round-Robin is also used by some authors to refer to the special case of TDMA arbitration in which the bus schedule period is partitioned into $|\mathit{Cores}|$ time slots of equal size and each core is assigned exactly one time slot [Hong et al., 2010]. In this thesis, in contrast, we only use the term Round-Robin to refer to the event-driven bus arbitration policy described above.

7.4. A Co-Runner-Insensitive Analysis

In this section, we derive a *co-runner-insensitive* WCET analysis for program *prog* executed on processor core C_i. This means that the analysis does not make any assumptions about the programs executed on the concurrent cores. Thus, this corresponds to the *Murphy* approach (cf. Section 2.2) of implicitly assuming maximally interfering programs on the concurrent cores.

We start the derivation from the baseline abstract model $\widehat{\mathit{ExecRuns}}^{C_i}_{\mathit{prog},C_i}$ presented in Section 7.2. We use a lifted version of system property P_{rr} for the detection of infeasible abstract traces. Intuitively, property P_{rr} is co-runner-insensitive as it upper-bounds the number of blocked cycles experienced by core C_i independently of the programs on the concurrent cores. Property P_{rr}

is formally lifted to $\widehat{ExecRuns}^{C_i}_{prog,C_i}$ in the following way—fulfilling the soundness criterion (4.C1) for lifted properties.

$$\begin{aligned}
&[\exists t \in \gamma_{trace}(\widehat{t^{C_i}}) : P_{rr}(t)] \\
\underset{(7.13)}{\Rightarrow}\ &[\exists t \in \gamma_{trace}(\widehat{t^{C_i}}) : \forall acc \in getAccesses_{C_i}(t) : \left|getBlocked_{C_i}(t, acc)\right| \leq (|Cores| - 1) \cdot LAT] \\
\underset{(7.7)}{\Leftrightarrow}\ &[\forall acc \in getAccesses_{C_i}(\widehat{t^{C_i}}) : \left|getBlocked_{C_i}(\widehat{t^{C_i}}, acc)\right| \leq (|Cores| - 1) \cdot LAT] \\
\Leftrightarrow:\ &\widehat{P^{C_i}_{rr}}(\widehat{t^{C_i}})
\end{aligned} \tag{7.15}$$

We use the lifted property $\widehat{P^{C_i}_{rr}}$ for the detection of infeasible abstract traces during micro-architectural analysis. Intuitively, during the non-deterministic modeling of the bus access behavior (cf. Figure 7.3), we only have to consider those successors that do not exceed the access-local upper bound on the number of blocked cycles. Due to the soundness of property lifting, the resulting abstract model $\widehat{ExecRuns}^{C_i\prime}_{prog,C_i}$ is also sound with respect to the concrete traces corresponding to the program execution.

$$\widehat{ExecRuns}^{C_i\prime}_{prog,C_i} = \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i}_{prog,C_i} \mid \widehat{P^{C_i}_{rr}}(\widehat{t^{C_i}})\} \tag{7.16}$$

Thus, we can use it again as a baseline abstract model for the calculation of a WCET bound as presented in Chapter 6. In particular, we can use the lifted versions of further program properties from set $Prop_{prog,C_i}$ (cf. equation (6.16)) in order to improve the precision of the obtained WCET bound.

$$\widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} = \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime}_{prog,C_i} \mid \forall P_k \in Prop_{prog,C_i} : \widehat{P^{C_i}_k}(\widehat{t^{C_i}})\} \tag{7.17}$$

The lifted property $\widehat{P^{C_i}_{rr}}$ enables the calculation of finite WCET bounds for programs that are guaranteed to terminate. Moreover, it guarantees the termination of our implementation of the micro-architectural analysis as it does no longer consider an infinite amount of abstract successor states during the modeling of the bus access behavior (cf. Section 7.2). As a consequence, it also guarantees that assumptions (6.139) and (6.140) hold for the detailed graph, which represents the results of the micro-architectural analysis.

A co-runner-insensitive analysis implicitly considers maximally interfering programs on the concurrent processor cores. This, however, can be very pessimistic with respect to the programs actually executed on the concurrent cores. Thus, we also present co-runner-sensitive WCET analyses.

7.5. Co-Runner-Sensitive Analyses

One Abstract Model per Processor Core In this section, we derive *co-runner-sensitive* WCET analyses for a program *prog* executed on processor core C_i. Thus, in addition to the abstract model $\widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i}$ (cf. Section 7.4), it also considers a corresponding abstract model $\widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i}$ per concurrent processor core C_j—modeling the operation of core C_j while core C_i executes program *prog*.

Note that each abstract model only has precise knowledge about the events which occur on the core that it models explicitly. Abstract model $\widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i}$ has e.g. no knowledge about when a concurrent core C_j is granted access to the bus. Thus, it pessimistically assumes that core C_j might be granted access at any instant (i.e. the corresponding may-event always occurs).

However, it can never be sure that core C_j is granted access at a particular instant (i.e. the corresponding must-event never occurs).

$$\begin{aligned}&\forall \widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i\,''}_{prog,C_i} : \forall x \in \mathbb{N}_{<len(\widehat{t^{C_i}})} : \forall C_j \in \mathit{Cores} \setminus \{C_i\} : \\ &\widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_i}}, x) \wedge \neg \widehat{\mathit{Granted}}^{LB}_{C_j}(\widehat{t^{C_i}}, x)\end{aligned} \tag{7.18}$$

In the same way, the abstract model $\widehat{\mathit{ExecRuns}}^{C_j\,''}_{prog,C_i}$ of a concurrent core C_j has no knowledge about when the execution of program *prog* on core C_i starts respectively ends.

$$\begin{aligned}&\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \forall \widehat{t^{C_j}} \in \widehat{\mathit{ExecRuns}}^{C_j\,''}_{prog,C_i} : \forall x \in \mathbb{N}_{<len(\widehat{t^{C_j}})} : \\ &\widehat{\mathit{Start}}^{UB}_{prog,C_i}(\widehat{t^{C_j}}, x) \wedge \neg \widehat{\mathit{Start}}^{LB}_{prog,C_i}(\widehat{t^{C_j}}, x) \wedge \\ &\widehat{\mathit{End}}^{UB}_{prog,C_i}(\widehat{t^{C_j}}, x) \wedge \neg \widehat{\mathit{End}}^{LB}_{prog,C_i}(\widehat{t^{C_j}}, x)\end{aligned} \tag{7.19}$$

A Compound Abstract Model We exploit this complementary information in the different abstract models by combining them to a *compound abstract model* (cf. Section 4.2.1). A compound abstract trace has one component abstract trace per processor core.

$$\widehat{\mathit{Cmpnd}} = \widehat{\mathit{ExecRuns}}^{C_1\,''}_{prog,C_i} \times \widehat{\mathit{ExecRuns}}^{C_2\,''}_{prog,C_i} \times \ldots \tag{7.20}$$

A compound abstract trace only describes those concrete traces that all of its component abstract traces describe.

$$\forall \widehat{t} \in \widehat{\mathit{Cmpnd}} : \gamma_{trace,prog,C_i}(\widehat{t}) = \bigcap_{C_k \in \mathit{Cores}} \gamma^{C_k}_{trace,prog,C_i}(\pi^{C_k}_{trace}(\widehat{t})) \tag{7.21}$$

As a consequence of each component abstract trace only describing concrete traces of its own length (cf. equation (5.32)), all compound abstract traces with components of different lengths are guaranteed to be infeasible. Thus, the subset of all compound abstract traces in which all components have the same length also forms an abstract model of the concrete traces.

$$\widehat{\mathit{Cmpnd}'} = \{\widehat{t} \in \widehat{\mathit{Cmpnd}} \mid \exists n \in \mathbb{N} : \forall C_k \in \mathit{Cores} : len(\pi^{C_k}_{trace}(\widehat{t})) = n\} \tag{7.22}$$

The length of a compound abstract trace in $\widehat{\mathit{Cmpnd}'}$ shall be defined as the common length of all its component abstract traces.

$$\forall \widehat{t} \in \widehat{\mathit{Cmpnd}'} : \forall C_k \in \mathit{Cores} : len(\pi^{C_k}_{trace}(\widehat{t})) = len(\widehat{t}) \tag{7.23}$$

A may-event (must-event) at a particular position of a compound abstract trace in $\widehat{\mathit{Cmpnd}'}$ is defined as the most precise may-event (must-event) at this position over all component abstract traces.

$$\begin{aligned}&\forall \widehat{t} \in \widehat{\mathit{Cmpnd}'} : \forall x \in \mathbb{N}_{<len(\widehat{t})} : \forall E \in \mathit{Events} : \\ &\quad \widehat{E^{UB}}(\widehat{t}, x) = \min_{C_k \in \mathit{Cores}} \widehat{E^{UB}}(\pi^{C_k}_{trace}(\widehat{t}), x) \wedge \\ &\quad \widehat{E^{LB}}(\widehat{t}, x) = \max_{C_k \in \mathit{Cores}} \widehat{E^{LB}}(\pi^{C_k}_{trace}(\widehat{t}), x)\end{aligned} \tag{7.24}$$

This means that compound abstract traces from $\widehat{\mathit{Cmpnd}'}$ provide the same interface to lifted properties as their component abstract traces. As a consequence, we can reuse the lifting rules of Section 5.2 for lifting system properties to the compound abstract model $(\widehat{\mathit{Cmpnd}'}, \gamma_{trace,prog,C_i})$.

The compound abstract model $(\widehat{Cmpnd}', \gamma_{trace,prog,C_i})$ argues about all processor cores. Thus, we can effectively use lifted versions of system properties interrelating the operation of the different cores for the detection of infeasible compound abstract traces. For an abstract model only arguing about one core, in contrast, such lifted system properties are essentially useless since the model does not argue about the other cores and, thus, has to make maximally pessimistic assumptions about their operation.

The *work-conserving* property (cf. equation (7.14)) is an example for a system property that interrelates the operation of the different processor cores. It can safely be lifted to the abstract model given by the compound abstract traces—resulting in the following lifted property $\widehat{P_{wc}}$.

$$\widehat{P_{wc}}(\hat{t}) \Leftrightarrow \forall x \in \mathbb{N}_{\leq len(\hat{t})} : \sum_{y\in\mathbb{N}_{<x}} \widehat{Blocked}^{LB}_{C_i}(\hat{t},y) \leq \sum_{C_j\in Cores\setminus\{C_i\}} \sum_{y\in\mathbb{N}_{<x}} \widehat{Granted}^{UB}_{C_j}(\hat{t},y) \tag{7.25}$$

Intuitively, this states that the number of cycles during which core C_i is guaranteed to be blocked at the shared bus is upper-bounded by the number of cycles during which a concurrent core may be granted access to the bus. For a formal proof that the lifted version $\widehat{P_{wc}}$ fulfills soundness criterion (4.C1) with respect to the system property P_{wc}, we refer to page 298.

Thus, the lifted property $\widehat{P_{wc}}$ can safely be used for the detection of infeasible compound abstract traces. The remaining set $\widehat{Cmpnd}''$ provides a sound overapproximation of the concrete traces.

$$\widehat{Cmpnd}'' = \{\hat{t} \in \widehat{Cmpnd}' \mid \widehat{P_{wc}}(\hat{t})\} \tag{7.26}$$

Iterative Overapproximation Starting from a Pessimistic Initialization However, the cross product in the definition of the set of compound abstract traces (cf. equation (7.20)) indicates that this compound consideration is prohibitive in terms of analysis runtime and/or memory consumption. Thus, we resort to a *processor-core-modular* overapproximation of $\widehat{Cmpnd}''$ following the iterative approach presented in Section 4.2.3.

The iterative overapproximation uses one approximation variable per component abstract model and initializes it *pessimistically*. Recall that $\widehat{ExecRuns}^{C_k{}''}_{prog,C_i}$ is the abstract trace set modeling only core C_k (cf. Section 7.4).

$$\forall C_k \in Cores : \widehat{Approx}^{C_k} \leftarrow \widehat{ExecRuns}^{C_k{}''}_{prog,C_i} \tag{7.27}$$

Subsequently, the approximation variables are updated in the following way.

$$\begin{aligned} &\forall C_j \in Cores \setminus \{C_i\} : \\ &\quad \widehat{Approx}^{C_j} \leftarrow \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j{}''}_{prog,C_i} \mid \\ &\qquad \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle}^{LB}(\widehat{t^{C_j}},x) \leq \max_{\widehat{t^{C_i}}\in\widehat{Approx}^{C_i}} \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle}^{UB}(\widehat{t^{C_i}},x)\} \end{aligned} \tag{7.28}$$

$$\begin{aligned} &\widehat{Approx}^{C_i} \leftarrow \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i{}''}_{prog,C_i} \mid \\ &\quad \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}},x) \leq \sum_{C_j\in Cores\setminus\{C_i\}} \max_{\widehat{t^{C_j}}\in\widehat{Approx}^{C_j}} \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}},x)\} \end{aligned} \tag{7.29}$$

The intuitive meaning of these updates gets more clear if we factor out the maxima and present the result as an iterative algorithm. This is demonstrated in Algorithm 7.1. First, it calculates a WCET bound for program *prog* executed on core C_i based on the current value of

$\widehat{Approx}^{C_i}$. Subsequently, for each concurrent processor core C_j, it calculates an upper bound on the number of granted access cycles that processor core C_j can produce in any interval not longer than $WCET^{UB}_{prog,C_i}$ clock cycles. This can be seen as the calculation of a value on an *arrival curve* for core C_j. Finally, the approximation variable of core C_i is updated in order to not consider any abstract traces that are guaranteed to be blocked longer than the sum over the arrival curve values of the concurrent cores. This sequence of steps is repeated by the main loop of the iterative algorithm. Note that actual implementations of the iterative algorithm feature additional machinery for the detection of a fixed point [Jacobs et al., 2015]. For the sake of readability, such machinery is omitted in Algorithm 7.1.

Algorithm 7.1 : Iterative WCET analysis following equations (7.28) and (7.29). It relies on a *maximally pessimistic initialization* (cf. equation (7.27)) of the approximation variables.

```
begin
    repeat
        $WCET^{UB}_{prog,C_i} \longleftarrow \max_{\widehat{t^{C_i}} \in \widehat{Approx}^{C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle}^{UB}(\widehat{t^{C_i}}, x)$
        for $C_j \in Cores \setminus \{C_i\}$ do
            $\widehat{Approx}^{C_j} \longleftarrow \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i} \mid$
                $\sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle}^{LB}(\widehat{t^{C_j}}, x) \leq WCET^{UB}_{prog,C_i}\}$
            $Grant^{UB}_{C_j} \longleftarrow \max_{\widehat{t^{C_j}} \in \widehat{Approx}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x)$
        $\widehat{Approx}^{C_i} \longleftarrow \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} \mid$
            $\sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in Cores \setminus \{C_i\}} Grant^{UB}_{C_j}\}$
    until ...
```

On page 299, we prove that the underlying formal setup (i.e. equations (7.27), (7.28), and (7.29)) fulfills all criteria required by Section 4.2.3. As a consequence, the resulting analysis approach is safe and monotone. It is an *anytime algorithm.* Thus, any WCET bound it calculates is sound (i.e. it is safe to choose an arbitrary termination condition for the main loop of Algorithm 7.1). Due to the monotonicity and the maximally pessimistic initialization, recalculations of the WCET bound can never lead to less precise values than earlier calculations. The algorithm should be stopped at latest when it reaches a fixed point as this means that continuing it would not lead to further improvements of the precision anyway.

Our Actual Implementation In this section, we derived an iterative WCET analysis based on one abstract model at the level of approximation of sequences of abstract states per processor core. Our actual implementation [Jacobs et al., 2015], which is also evaluated in the implementation part of this thesis, does not directly perform the bound calculations at the level of approximation of sequences of abstract states. Instead, it performs them at the higher level of approximation of implicit path enumeration (cf. Section 5.4). The additional constraints on the abstract traces—in the updates (cf. equations (7.28) and (7.29)) of the approximation variables—are lifted to the level of approximation of implicit path enumeration. Thus, the contents of the approximation variables are never explicitly considered in our implementation of the iterative approach.

For the calculations of the WCET bound $\mathit{WCET}^{UB}_{prog,C_i}$ in the implementation of the iterative approach, we can directly reuse the techniques presented in Chapter 6. Intuitively, we precisely model the execution of program *prog* on processor core C_i and—based on the results—calculate an upper bound on the execution times.

For the calculations of the arrival curve values $\mathit{Grant}^{UB}_{C_j}$, in contrast, the techniques of Chapter 6 are not directly applicable. Intuitively, we need to calculate an upper bound on the number of granted bus access cycles that core C_j can perform while core C_i executes program *prog*. Thus, we might have to consider sequences of more than only a single program execution run on core C_j. The techniques of Chapter 6, however, calculate an upper bound on the number of bus access cycles granted to core C_j during any program execution run of a particular program executed on core C_j. Thus, in Chapter 10, we present multiple techniques for the calculation of arrival curve values for core C_j—differing in precision and computational complexity. Note that the contents of Chapter 10 are presented at a less formal level. A similarly formal derivation as in Chapter 6 would—in principle—have been possible, but is omitted due to time and space constraints.

Iterative Overapproximation Starting from an Optimistic Initialization When defining the iterative overapproximation approach starting from a pessimistic initialization, we also experimented with using the same approach in combination with an obvious optimistic initialization. However, it turned out that this combination does not fulfill all criteria required by Section 4.2.4. From page 302 onward, we present a counter example demonstrating the unsoundness and discuss an additional but typically undesired assumption under which such an approach is sound.

In order to overcome the soundness problems of our first attempt with an optimistic initialization, we propose a slight variant of this first attempt. The variant initializes and updates the approximation variable of core C_i in a slightly different way than in our first attempt. The initialization of the approximation variables shall be optimistically chosen as follows. Note that the optimistic initialization of the approximation variable of core C_i permits one guaranteed blocked cycle per abstract trace—but only at the respective tail position (i.e. at position $\mathit{len}(\widehat{t^{C_i}}) - 1$).

$$\widehat{\mathit{Approx}}^{C_i} \leftarrow \{\widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\} \tag{7.30}$$

$$\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \widehat{\mathit{Approx}}^{C_j} \leftarrow \{\widehat{t^{C_j}} \in \widehat{\mathit{ExecRuns}}^{C_j\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_j}})}} \widehat{\mathit{Cycle}}^{LB}(\widehat{t^{C_j}}, x) \leq 0\} \tag{7.31}$$

Analogously, the update of the approximation variable of core C_i only upper-bounds the number of guaranteed blocked cycles at all non-tail positions. Note that the updates of all other approximation variables are identical to the corresponding updates of the approach starting from a pessimistic initialization (cf. equation (7.28)).

$$\begin{aligned}\widehat{\mathit{Approx}}^{C_i} \leftarrow \{&\widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} \mid \\ &\sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in \mathit{Cores} \setminus \{C_i\}} \max_{\widehat{t^{C_j}} \in \widehat{\mathit{Approx}}^{C_j}} \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x)\}\end{aligned} \tag{7.32}$$

For the moment, the following additional assumption shall hold. It states that the set $\widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i}$ shall contain all prefixes of each of its members. Intuitively, this means that we must not use lifted properties that detect the infeasibility of a given abstract trace but do not detect the infeasibility of one of its extensions.

$$\begin{aligned} &\forall \widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} : \\ &\quad len(\widehat{t^{C_i}}) \geq 1 \\ &\quad \Rightarrow \exists \widehat{t^{C_i\prime}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} : \\ &\qquad (\widehat{t^{C_i\prime}}, \widehat{t^{C_i}}) \in \widehat{PrefixOf} \wedge \\ &\qquad len(\widehat{t^{C_i\prime}}) = len(\widehat{t^{C_i}}) - 1 \end{aligned} \tag{7.33}$$

Under assumption (7.33), the formal setup of the proposed approach (i.e. equations (7.30), (7.31), (7.28), and (7.32)) is guaranteed to fulfill all criteria required by Section 4.2.4. For a formal proof of this, we refer to page 305. Algorithm 7.2 corresponds to the fixed point iteration of this proposed approach.

Algorithm 7.2 : Iterative WCET analysis following equations (7.28) and (7.32). It relies on an *optimistic initialization* following equations (7.30) and (7.31).

```
begin
    repeat
        $WCET^{UB}_{prog,C_i} \longleftarrow \max_{\widehat{t^{C_i}} \in \widehat{Approx}^{C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle}^{UB}(\widehat{t^{C_i}}, x)$
        for $C_j \in Cores \setminus \{C_i\}$ do
            $\widehat{Approx}^{C_j} \longleftarrow \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i} \mid$
                $\sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle}^{LB}(\widehat{t^{C_j}}, x) \leq WCET^{UB}_{prog,C_i}\}$
            $Grant^{UB}_{C_j} \longleftarrow \max_{\widehat{t^{C_j}} \in \widehat{Approx}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x)$
        $\widehat{Approx}^{C_i} \longleftarrow \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} \mid$
            $\sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in Cores \setminus \{C_i\}} Grant^{UB}_{C_j}\}$
    until fixed point reached
```

Even if assumption (7.33) does not hold for all members of $\widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i}$, it holds by definition for all feasible members of $\widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i}$, i.e. any prefix of a feasible sequence is again feasible and lifted properties never prune feasible stuff. Thus, intuitively, the fixed point is guaranteed to safely overapproximate all feasible members of the projections of the compound consideration. A formal proof of this, however, exceeds the scope of this thesis. As the feasible members of a projection safely overapproximate the concrete traces, the fixed point does so as well. Thus, the WCET bound at the fixed point is sound with respect to the concrete system—even if assumption (7.33) does not hold.

Note that, for the considered system with Round-Robin bus arbitration, Algorithm 7.2 is guaranteed to reach a fixed point after a finite number of iterations. There are essentially two cases that can happen:

- If the co-runner-insensitive WCET bound is already undefined, Algorithm 7.2 also immediately reaches an undefined value for the WCET bound because shared-bus interference cannot be the reason for the undefined WCET bound anyway (due to property P_{rr}). Thus, independently of the calculated arrival curve values, the WCET bound won't improve in the next iteration and a fixed point will be reached.

- If the co-runner-insensitive WCET bound has a defined value, Algorithm 7.2 will at latest terminate when the WCET bound reaches this finite value. Due to monotonicity, only a finite number of iterations will be needed for this.

For arbitrary systems with work-conserving bus arbitration, however, Algorithm 7.2 might diverge. Thus, if we cannot guarantee its termination for a particular system, we should extend Algorithm 7.2 by additional machinery that stops the iteration as soon as the goal of the overall timing verification cannot be shown anymore (e.g. as soon as the current WCET bound already exceeds the corresponding deadline).

Implementing the Approach Starting from an Optimistic Initialization Due to time constraints, we have not implemented the approach starting from an optimistic initialization. Nonetheless, we sketch a possible implementation and some of its implications.

In our actual implementation of the micro-architectural analysis, we directly use the lifted property $\widehat{P_{rr}^{C_i}}$ for the pruning of infeasible abstract successor states (cf. equation (7.16)). In the same way, we also use some of the lifted versions $\widehat{P_k^{C_i}}$ of other system properties (cf. equation (7.17)) directly during micro-architectural analysis. Some lifted properties $\widehat{P_k^{C_i}}$ (typically the cumulative ones), however, are not used during micro-architectural analysis. Instead, they are further lifted to the higher levels of approximation. Thus, the results of the micro-architectural analysis of core C_i ($\widehat{\mathit{ExecRuns}}^{C_i,\mu Arch}_{prog,C_i}$) are between the two sets that we considered during the derivation of our co-runner-insensitive analysis (cf. Section 7.4) with respect to the subset relation.

$$\widehat{\mathit{ExecRuns}}^{C_i\prime}_{prog,C_i} \supseteq \widehat{\mathit{ExecRuns}}^{C_i,\mu Arch}_{prog,C_i} \supseteq \widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} \tag{7.34}$$

$$\widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} \underset{\substack{(7.17)\\(7.34)}}{=} \{\widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i,\mu Arch}_{prog,C_i} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k^{C_i}}(\widehat{t^{C_i}})\} \tag{7.35}$$

Let G^B be a graph obtained from $\widehat{\mathit{ExecRuns}}^{C_i,\mu Arch}_{prog,C_i}$ by applying the graph construction proposed in Section 6.4.4 and optionally also a sequence of subsequent graph transformations (cf. Section 6.4.6). Let *Prefixes* denote the set of prefixes of paths through G^B for which the lifted versions of all system properties hold.

$$\mathit{Prefixes} = \{\widehat{p} \in \widehat{\mathit{RelPaths}}^B \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k^{path}}(\widehat{t^{C_i}})\} \tag{7.36}$$

Based on these prefixes, we can provide a drop-in replacement for the WCET bound calculations in Algorithm 7.2. To this end, we initialize the approximation variable of core C_i to the path prefixes in which at most the last edge may guarantee blocked cycles.

$$\widehat{\mathit{Approx}}^{C_i} \leftarrow \{\widehat{p} \in \mathit{Prefixes} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{p})-1}} \widehat{\mathit{wBlocked}}^{LB}_{C_i}(\widehat{p}, x) \leq 0\} \tag{7.37}$$

Analogously, the update of the approximation variable of core C_i in Algorithm 7.2 is replaced by a corresponding update of the path prefixes.

$$\widehat{Approx}^{C_i} \leftarrow \{\widehat{p} \in \mathit{Prefixes} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{p})-1}} \widehat{wBlocked}^{LB}_{C_i}(\widehat{p}, x) \leq \sum_{C_j \in Cores\setminus\{C_i\}} Grant^{UB}_{C_j}\} \tag{7.38}$$

Finally, the actual WCET bound calculation is also replaced by a corresponding calculation on path prefixes.

$$WCET^{UB}_{prog,C_i} \longleftarrow \max_{\widehat{p} \in \widehat{Approx}^{C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCycle}^{UB}(\widehat{p}, x) \tag{7.39}$$

Intuitively, the value of the WCET bound at the fixed point is sound with respect to the concrete system. A formal soundness proof, however, is omitted due to space and time constraints.

Note that, in order to guarantee the soundness, we have to argue about all path prefixes of graph G^B (cf. equation (7.36)). As implicit path enumeration, however, only argues about all paths from the start to the end of a graph (cf. Section 5.4), we perform it on a graph $G^{B\prime}$ which is obtained from G^B by reinterpreting all nodes as end nodes.

In the same way as for the iterative approach starting from a pessimistic initialization, the calculation of values on arrival curves shall be based on the techniques presented in Chapter 10.

Reusing the Fixed Point Obtained from an Optimistic Initialization as Pessimistic Initialization The fixed point of Algorithm 7.2 is sound with respect to the concrete system. Thus, we can safely reuse this fixed point as pessimistic initialization for Algorithm 7.1. In this way, we can potentially improve the precision of the WCET bound once Algorithm 7.2 has reached a fixed point.

Bounding the Blocked Cycles More Precisely under Round-Robin Bus Arbitration So far, we only exploit a property of Round-Robin bus arbitration that bounds the overall amount of cycles that a shared-bus access of a processor core can be blocked (P_{rr}, cf. equation (7.13)). The co-runner-sensitive analyses, in contrast, only relies on the fact that the bus arbitration policy is work-conserving (P_{wc}, cf. equation (7.14)). As a consequence, in Algorithm 7.1 and Algorithm 7.2, the arrival curve values of the concurrent processor cores are simply added up in order to obtain an upper bound on the number of blocked cycles.

$$\ldots \leq \sum_{C_j \in Cores\setminus\{C_i\}} Grant^{UB}_{C_j}$$

We can improve the precision of the co-runner-sensitive analyses by making them aware of an even stronger property than P_{wc}. Intuitively, each requested bus access of core C_i can at most be blocked for LAT cycles due to a particular concurrent processor core C_j. Still, the overall amount of cycles that the program execution on core C_i is blocked due to core C_j must not exceed the overall amount of cycles core C_j is granted access to the bus. This results in the following additional property of Round-Robin bus arbitration.

$$\begin{aligned} P_{rr'}(t) \Leftrightarrow & \forall x \in \mathbb{N}_{\leq len(t)} : \\ & \sum_{y \in \mathbb{N}_{<x}} Blocked_{C_i}(t,y) \leq \sum_{C_j \in Cores\setminus\{C_i\}} \min[\sum_{y \in \mathbb{N}_{<x}} Granted_{C_j}(t,y), \\ & \qquad \sum_{y \in \mathbb{N}_{<x}} Requested_{C_i}(t,y) \cdot LAT\,] \end{aligned} \tag{7.40}$$

Relying on this property, we can improve the upper bound on the number of blocked cycles used in Algorithm 7.1 and Algorithm 7.2 as follows.

$$\ldots \leq \sum_{C_j \in Cores \setminus \{C_i\}} \min[\, Grant^{UB}_{C_j}, \sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_i}})}} \widehat{Requested}^{UB}_{C_i}(\widehat{t^{C_i}}, x) \cdot LAT \,]$$

Since one operand of the minimum expression depends on the abstract trace $\widehat{t^{C_i}}$, we cannot statically evaluate its value before the update of the approximation variable. In case the implementing technology at a higher level of approximation (e.g. an ILP-based implicit path enumeration) does not directly support minimum expressions in linear constraints, we can simulate the minimum expressions by additional integer variables and linear constraints as follows.

$$\begin{aligned}
&\ldots \leq \sum_{C_j \in Cores \setminus \{C_i\}} BlockedBy_{C_j} \;\wedge \\
&\forall C_j \in Cores \setminus \{C_i\} : \\
&\quad BlockedBy_{C_j} \leq Grant^{UB}_{C_j} \;\wedge \\
&\quad BlockedBy_{C_j} \leq \sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_i}})}} \widehat{Requested}^{UB}_{C_i}(\widehat{t^{C_i}}, x) \cdot LAT
\end{aligned}$$

Alternatively, we can upper bound the operand of the minimum expression which depends on the abstract trace $\widehat{t^{C_i}}$ by the corresponding maximum over all possible abstract traces. In this way, the minimum expression can be statically evaluated before the update of the approximation variable. This effectively reduces the complexity of every WCET bound calculation.

$$MaxBlockedPerConcCore = \max_{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i''}_{prog,C_i}} \sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_i}})}} \widehat{Requested}^{UB}_{C_i}(\widehat{t^{C_i}}, x) \cdot LAT$$

$$\ldots \leq \sum_{C_j \in Cores \setminus \{C_i\}} \min[\, Grant^{UB}_{C_j}, MaxBlockedPerConcCore \,]$$

In the actual implementation of Algorithm 7.1, we choose this last variant of statically calculating *MaxBlockedPerConcCore* only once [Jacobs et al., 2015]. Note, however, that this variant can potentially be less precise than the variant presented before in case the abstract traces maximizing the number of access requests do not coincide with the abstract traces maximizing the execution time.

Further note that the presented methods for bounding the number of blocked cycles more precisely under Round-Robin bus arbitration could be formally derived in a similar way as the vanilla versions of Algorithm 7.1 and Algorithm 7.2. Such a formal derivation, however, is omitted due to space and time constraints.

7.6. Quantifying Shared-Bus Interference of a Concurrent Core: Granted Access Cycles vs. Granted Accesses

The co-runner-sensitive WCET analyses presented in Section 7.5 quantify the shared-bus interference generated by the concurrent processor cores at the *granularity of granted access cycles*. The iterative approaches calculate an upper bound on the number of granted access cycles per concurrent core. During the subsequent recalculation of the WCET bound, the approaches only consider abstract traces whose number of blocked cycles does not exceed the sum of the upper bounds on the granted access cycles of the concurrent cores. Intuitively, the WCET bound calculation distributes the assumed budget of concurrently granted access cycles among the accesses of the core under analysis in a way that the WCET bound is maximized.

Potential Imprecision of the Presented Co-Runner-Sensitive Analyses On the concrete system, however, the granted access cycles of the concurrent processor cores cannot be distributed among all accesses of the core under analysis in a completely free manner. The granted cycles of a particular access of a concurrent processor core can only block a single access of the core under analysis. As a consequence, quantifying the concurrent shared-bus interference at the granularity of granted access cycles (cf. Section 7.5) might lead to imprecision as the worst-case considered at the abstract model might be infeasible with respect to the concrete traces.

In this section, we propose to avoid such imprecision by alternatively quantifying the shared-bus interference generated by a concurrent processor core at the *granularity of granted accesses*. Due to space and time constraints, however, we only sketch the application of this alternative granularity and omit the corresponding formal derivation. Nonetheless, we try to provide an intuitive soundness argument by beginning the presentation with the two key properties that this alternative approach relies on.

The first key property states that any access of processor core C_i can at most be blocked for LAT cycles per concurrent access that is granted while the access of core C_i is pending.

$$\begin{aligned}
&P_{AccGranu_1}(t)\\
\Leftrightarrow&\forall acc_{C_i} \in getAccesses_{C_i}(t) :\\
&\quad \left|getBlocked_{C_i}(t, acc_{C_i})\right| \leq LAT \cdot \Big|\{acc_{C_j} \in getAccesses_{C_j}(t) \mid C_j \in Cores \setminus \{C_i\} \land\\
&\qquad\qquad getPending_{C_i}(t, acc_{C_i}) \cap getGranted_{C_j}(t, acc_{C_j}) \neq \emptyset\}\Big|
\end{aligned}$$

The second key property states that, for any access of a concurrent core C_j, there is at most one access of core C_i pending while the access of core C_j is granted.

$$\begin{aligned}
&P_{AccGranu_2}(t)\\
\Leftrightarrow&\forall C_j \in Cores \setminus \{C_i\} : \forall acc_{C_j} \in getAccesses_{C_j}(t) :\\
&\quad \left|\{acc_{C_i} \in getAccesses_{C_i}(t) \mid getPending_{C_i}(t, acc_{C_i}) \cap getGranted_{C_j}(t, acc_{C_j}) \neq \emptyset\}\right| \leq 1
\end{aligned}$$

The intuitive consequence of these properties is that every granted access of a concurrent core C_j can block at most one access of core C_i and at most for LAT cycles.

Co-Runner-Sensitive Analyses Quantifying Shared-Bus Interference at the Granularity of Granted Accesses We can exploit this for the design of co-runner-sensitive analyses similar to those presented in Section 7.5. This time, however, we distribute the budget of granted accesses of the concurrent cores C_j among the accesses of core C_i in a way that the WCET bound is maximized.

Intuitively, the budget distribution works as follows.

- If an access of core C_i is not blocked at all, it does not consume any granted accesses from the budget.
- If an access of core C_i is blocked between 1 and LAT cycles, it consumes 1 granted access from the budget.
- If an access of core C_i is blocked between $LAT + 1$ and $2 \cdot LAT$ cycles, it consumes 2 granted accesses from the budget.
- ...
- More general: if an access of core C_i is blocked between $(x - 1) \cdot LAT + 1$ and $x \cdot LAT$ cycles, it consumes x granted accesses from the budget.

In order to express this in an efficient way during our non-deterministic modeling of the shared-bus interference (cf. Section 7.2), we introduce the pseudo-event $\mathit{ConcGrantAcc}_{C_i}$. It occurs if and only if core C_i is blocked at the shared bus and the number of blocked cycles since the corresponding access request equals to 1 modulo LAT.

$$\begin{aligned}\mathit{ConcGrantAcc}_{C_i}(t, x) \Leftrightarrow & \mathit{Blocked}_{C_i}(t, x) \wedge \\ & \sum_{y \in \mathbb{N}_{\leq x} \cap \mathbb{N}_{\geq \max\{z \in \mathbb{N}_{\leq x} \mid \mathit{Requested}_{C_i}(t,z)\}}} \mathit{Blocked}_{C_i}(t, y) \equiv 1 \bmod LAT\end{aligned}$$

More practically spoken, this means that the pseudo-event occurs whenever an access of core C_i is blocked for the $1^{\text{st}}, (LAT + 1)^{\text{th}}, (2 \cdot LAT + 1)^{\text{th}}, \ldots$ time. Intuitively, the pseudo-event means: starting from the current blocked cycle, another granted access of a concurrent processor core is consumed. For a concrete value of LAT, we could blow up the diagram representation of the non-determinism (cf. Figure 7.3) accordingly in order to also describe the new pseudo-event in the diagram. For the sake of readability, we do not present such a blown-up diagram in this section. In our actual implementation of this non-determinism, we efficiently model the number of blocked cycles since the corresponding access request by a counter variable.

In the co-runner-sensitive analyses presented in Section 7.5 (Algorithm 7.1 and Algorithm 7.2), we change a few details in order to switch from granted access cycles to granted accesses. In the adapted versions of both algorithms, now, we calculate an upper bound on the number of granted accesses instead of an upper bound on the number of granted access cycles.

$$\mathit{Grant}_{C_j}^{UB} \longleftarrow \max_{\widehat{t^{C_j}} \in \widehat{\mathit{Approx}}^{C_j}} \left| \{ \mathit{acc}_{C_j} \in \mathit{getAccesses}_{C_j}(\widehat{t^{C_j}}) \mid \mathit{getGranted}_{C_j}(\widehat{t^{C_j}}, \mathit{acc}_{C_j}) \neq \emptyset \} \right|$$

In case it is too complicated to argue about the number of granted accesses at a higher level of approximation (e.g. during implicit path enumeration), we can safely upper-bound the above value by maximizing the number of requested accesses and subsequently adding one. Note that the author was initially not aware of this and has to thank Sebastian Hahn for pointing it out.

$$\mathit{Grant}_{C_j}^{UB} \longleftarrow 1 + \max_{\widehat{t^{C_j}} \in \widehat{\mathit{Approx}}^{C_j}} \sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_j}})}} \widehat{\mathit{Requested}}_{C_j}^{UB}(\widehat{t^{C_j}}, x)$$

In the adapted version of Algorithm 7.1, we use these upper bounds to constraint the number of occurrences of pseudo-event $\mathit{ConcGrantAcc}_{C_i}$ that can happen in an abstract trace.

$$\sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_i}})}} \widehat{\mathit{ConcGrantAcc}}_{C_i}^{LB}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in \mathit{Cores} \setminus \{C_i\}} \mathit{Grant}_{C_j}^{UB}$$

In the adapted version of Algorithm 7.2, we only constraint the number of occurrences of pseudo-event $ConcGrantAcc_{C_i}$ that can happen at all non-tail positions of an abstract traces.

$$\sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_i}})-1}} \widehat{ConcGrantAcc}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in Cores \setminus \{C_i\}} Grant^{UB}_{C_j}$$

Moreover, the optimistic initialization of Algorithm 7.2 is adapted accordingly.

$$\widehat{Approx}^{C_i} \leftarrow \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{< len(\widehat{t^{C_i}})-1}} \widehat{ConcGrantAcc}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\}$$

Note that we expect the soundness of the adapted version of Algorithm 7.2 to additionally rely on the following slight variant of property $P_{AccGranu_1}$. This is, however, only an educated guess based on our experiences with the soundness proof of the original version of Algorithm 7.2. A formal soundness proof of the adapted version of Algorithm 7.2 exceeds the scope of this thesis.

$$\begin{aligned}
&\Leftrightarrow P^{*}_{AccGranu_1}(t) \\
&\quad \forall acc_{C_i} \in getAccesses_{C_i}(t) : \\
&\quad\quad \forall x \in getBlocked_{C_i}(t, acc_{C_i}) : \\
&\quad\quad\quad \left|\mathbb{N}_{\geq acc_{C_i}} \cap \mathbb{N}_{\leq x}\right| \leq LAT \cdot \Big|\{acc_{C_j} \in getAccesses_{C_j}(t) \mid C_j \in Cores \setminus \{C_i\} \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\quad \mathbb{N}_{\geq acc_{C_i}} \cap \mathbb{N}_{\leq x} \cap getGranted_{C_j}(t, acc_{C_j}) \neq \emptyset\}\Big|
\end{aligned}$$

Further note that the co-runner-sensitive analyses quantifying the interference generated by the concurrent cores at the granularity of granted accesses—as just sketched—do not exploit that the bus is arbitrated following the Round-Robin policy. Thus, they can also be applied to systems with other work-conserving bus arbitration policies. In a similar way as for the analyses quantifying the interference at the granularity of granted access cycles, we can improve the analysis precision by additionally taking into account that—under Round-Robin bus arbitration—each access of core C_i can at most be blocked by one granted access per concurrent processor core. The necessary changes to the analyses are analogous to those discussed at the end of Section 7.5 and, thus, not further discussed in this section.

Orthogonality of the Granularities with Respect to Precision Next, we demonstrate that none of both granularities dominates the other with respect to precision. To this end, consider the example graph of Figure 7.5. The graph represents the results of a micro-architectural analysis of a simple example program and is used for the calculation of a co-runner-sensitive WCET bound. The concrete system for which the analysis was performed shall feature a dual-core processor and a shared bus with Round-Robin arbitration and a granted-access latency LAT of two cycles. The program performs two accesses to the shared bus of which each one can be blocked for up to LAT (i.e. two) cycles according to the Round-Robin policy.

In a *first interference scenario*, assume that the concurrent processor core can be granted up to two access cycles. Nonetheless, the concurrent processor core shall be granted at most one access (i.e. the two granted access cycles must belong to the same access). In case we use the number of granted access cycles of the concurrent core to bound the number of blocked cycles on the core under analysis, we end up with a WCET bound of eight cycles (along the path $a \to b \to d \to e \to g \to h \to j \to k \to m$). In case we use the number of granted accesses of the concurrent core to bound the number of consumed concurrent accesses of the core under analysis, we end up with a WCET bound of seven cycles (e.g. along the path $a \to b \to c \to d \to e \to g \to l \to m$). Thus, in this interference scenario, it is more beneficial in terms of precision to use the number of granted accesses for quantifying the interference generated by the concurrent core.

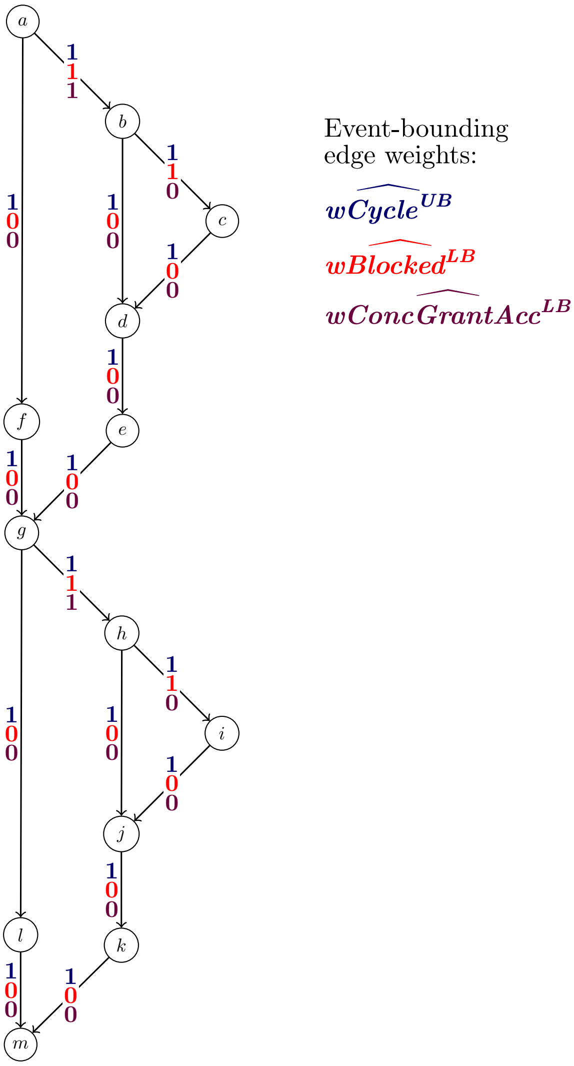


Figure 7.5.: Detailed graph of a simple example program. The edge weights upper-bound the number of **processor cycles**, lower-bound the number of **cycles blocked at the shared bus**, and lower-bound the number of **concurrently granted bus accesses consumed**.

In a *second interference scenario*, assume that the concurrent processor core can be granted up to three access cycles. Thus, the concurrent processor core can be granted up to two accesses (because a single granted access with $LAT = 2$ cannot result in three granted access cycles). In case we use the number of granted access cycles of the concurrent core to bound the number of blocked cycles on the core under analysis, we end up with a WCET bound of nine cycles (e.g. along the path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow g \rightarrow h \rightarrow j \rightarrow k \rightarrow m$). In case we use the number of granted accesses of the concurrent core to bound the number of consumed concurrent accesses of the core under analysis, we end up with a WCET bound of ten cycles (along the path $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow g \rightarrow h \rightarrow i \rightarrow j \rightarrow k \rightarrow m$). Thus, in this interference scenario, it is more beneficial in terms of precision to use the number of granted access cycles for quantifying the interference generated by the concurrent core.

This shows that none of both granularities dominates the other with respect to precision. As a consequence, one could think of combined iterative algorithms that rely on both granularities and, thus, calculate two arrival curve values per concurrent core and iteration. Such combined algorithms are a straight-forward extension of the iterative algorithms already presented or sketched. Thus, we do not further discuss them here.

Converting Interference Bounds from One Granularity to the Other As discussed before, one would need to calculate arrival curve values at both granularities in order to fully exploit the orthogonality of the granularities with respect to precision. If we can—for whatever reason—only calculate arrival curve values at one of the granularities, we can convert a value to the respective other granularity and potentially still profit to some degree from the orthogonality of both granularities.

The safe conversion from the granularity of granted accesses to the granularity of granted access cycles is performed as follows. Let x upper-bound the number of accesses that core C_j can be granted during any time interval of at most W cycles (and, thus, $x \leq W$). Then, $\mathit{toAccCycles}(W, x)$ upper-bounds the number of access cycles that core C_j can be granted during any time interval of at most W cycles. The conversion pessimistically assumes that any granted access is granted for LAT cycles. Moreover, it exploits that any time interval of at most W cycles can at most contain W granted access cycles.

$$\mathit{toAccCycles}(W, x) = \min(W, x \cdot \mathit{LAT}) \tag{7.41}$$

Note that the number of access cycles obtained from this conversion is guaranteed to not lead to an improvement of precision compared to the use of the initial number x of accesses unless $W < x \cdot \mathit{LAT}$. Thus, we expect that the application of this conversion does in most cases not lead to an improved precision.

The safe conversion from the granularity of granted access cycles to the granularity of granted accesses is performed as follows. Let x upper-bound the number of access cycles that core C_j can be granted during any time interval of at most W cycles (and, thus, $x \leq W$). Then, $\mathit{toAccesses}(x)$ upper-bounds the number of accesses that core C_j can be granted during any time interval of at most W cycles. The conversion upper-bounds to how many different granted accesses the x granted access cycles can at most belong.

$$\mathit{toAccesses}(x) = \min(x, \left\lceil \frac{x-1}{\mathit{LAT}} \right\rceil + 1) \tag{7.42}$$

The effectiveness of this conversion can be demonstrated with a simple example graph similar to the graph of Figure 7.5. The considered hardware platform shall also be the same as for the examples presented in combination with Figure 7.5. This time, however, the pattern which is repeated twice in Figure 7.5 is repeated five times. Thus, the resulting graph stands for a program which performs five bus accesses. Now, assume that the concurrent core is granted at most five access cycles. If we directly use these five access cycles to upper-bound the number of blocked cycles of the program execution, we obtain a WCET bound of twenty cycles. If we, however, convert the five access cycles into three granted accesses (following equation (7.42)) and use this to upper-bound the number of concurrently granted accesses consumed by the program execution, we obtain a WCET bound of nineteen cycles. Due to space restrictions, we do not further visualize this example.

Note that we do not formally prove the correctness of the presented conversion rules.

Use of the Granularities in Literature *Fully integrated* WCET analyses model the shared-bus interference (more or less) exactly by enumerating (an overapproximation of) all interleavings of accesses to the bus performed by the different processor cores [Lv et al., 2010; Gustavsson et al., 2010; Kelter and Marwedel, 2014; Giannopoulou et al., 2012; Lampka et al., 2014]. These approaches are not processor-core-modular and, thus, do not rely on cumulative approximations of the interference generated by the concurrent processor cores. Hence, we cannot classify these approaches at all in terms of the granularity at which they quantify the amount of shared-bus interference generated by a concurrent core. As discussed before (cf. Section 2.2.1), these approaches are unlikely to scale to real-world scenarios due to their enormous computational complexity.

Most existing approaches to timing verification for multi-core processors with shared buses avoid this complexity by taking a *processor-core-modular view* [Schliecker et al., 2008; Negrean et al., 2009; Andersson et al., 2010; Schliecker and Ernst, 2010; Pellizzoni et al., 2010; Schranzhofer et al., 2011; Dasari et al., 2011; Dasari and Nélis, 2012; Nowotsch and Paulitsch, 2013; Nowotsch et al., 2014; Altmeyer et al., 2015]. All of these processor-core-modular approaches quantify the shared-bus interference generated by the concurrent processor cores at the granularity of granted accesses.

To the best of our knowledge, we are first to point out the potential impact that the granularity at which the shared-bus interference generated by a concurrent core is quantified can have on the precision of WCET analysis. In particular, we presented co-runner-sensitive WCET analyses quantifying the interference generated by a concurrent core at the granularity of granted access cycles (cf. Section 7.5) and at the granularity of granted accesses (cf. this section). Subsequently, we demonstrated that—in general—none of both granularities dominates the other with respect to precision.

Due to space and time constraints, in the remainder of this thesis, we only consider co-runner-sensitive WCET analyses quantifying the interference generated by a concurrent core at the granularity of granted access cycles (cf. Section 7.5). Thus, in particular, we do not conduct an experimental case study on the impact that both granularities (or a combination of them) can have on the precision of WCET analysis.

7.7. Toward Priority-Based Bus Arbitration

Priority-based bus arbitration protocols base their arbitration decisions on priorities that are assigned to the different requests competing for access. In case multiple accesses are pending, the bus is granted to the core requesting access with the highest priority. As for all bus arbitration protocols we consider, once an access is granted, it stays granted until completion (cf. Figure 7.3). Note that priority-based bus arbitration is also *work-conserving* as it only blocks a processor in case another core is granted access to the bus.

There is a whole class of priority-based bus arbitration protocols. Its members differ in whether priorities are assigned to the processor cores, the programs, or potentially even to particular accesses. Moreover, they differ in whether the priorities are chosen statically (i.e. fixed-priority) or may change during the execution of the system. For the sake of simplicity, in this thesis, we only consider a priority-based bus arbitration protocol that uses a statically assigned, unique priority per processor core. This particular variant has also been referred to as *processor-priority* in literature [Altmeyer et al., 2015]. In the following, we also use this term.

Note that, for systems with processor-priority arbitration, programs executed on a processor core which is not assigned the highest priority might suffer from *starvation*. This means that a pending access of the core is never granted because there is continuously a pending access of a core with a higher priority. As a consequence, there are no finite co-runner-insensitive WCET bounds for programs executed on a processor core which is not assigned the highest priority.

Thus, in such scenarios, a co-runner-sensitive analysis is mandatory for the determination of finite WCET bounds. To this end, we can safely reuse the co-runner-sensitive WCET analyses presented in Section 7.5 since they only rely on a work-conserving bus arbitration (which the processor-priority arbitration happens to be). Note, however, that a co-runner-sensitive analysis starting from a maximally pessimistic initialization will also in most cases not lead to a finite WCET bound: The first WCET bound calculated starting from a pessimistic initialization is co-runner-insensitive and, thus, undefined for at least all programs executed on a core which is not assigned the highest priority. In an arbitrarily long interval of time (i.e. the interval length is not upper-bounded by a finite value), the concurrent cores can in nearly all cases produce an arbitrarily high amount of granted access cycles. Hence, a recalculation of the WCET bound again results in an undefined value, which corresponds to a fixed point of the iterative analysis. As a consequence, a co-runner-sensitive WCET analysis can in most cases only calculate a finite WCET bound if it starts from an optimistic initialization.

Further note that a co-runner-sensitive WCET analysis starting from an optimistic initialization can in general diverge (e.g. if the program under analysis suffers from starvation on the concrete system). In order to avoid the divergence of the co-runner-sensitive analysis, the iterative algorithm shall be stopped as soon as the overall goal of the timing verification can no longer be proven (e.g. as soon as a deadline is exceeded).

Finally, note that the co-runner-sensitive WCET analyses solely relying on a work-conserving bus arbitration (as presented in Section 7.5) can result in overly pessimistic analysis results if applied to a system with a processor-priority bus arbitration: They upper-bound the number of cycles during which the core under analysis is blocked by the overall amount of access cycles granted to the concurrent cores. On a system with processor-priority arbitration, however, an access of the core under analysis can at most be blocked by one core of lower priority (i.e. an access of lower priority has already been granted before the access request of the core under analysis has been issued). This is expressed by the following system property.

$$\begin{aligned}
P_{\textit{proc-prio}}(t) \Leftrightarrow & \forall x \in \mathbb{N}_{\leq len(t)} : \\
& \sum_{y \in \mathbb{N}_{<x}} \textit{Blocked}_{C_i}(t,y) \leq \sum_{C_j \in \textit{HigherThan}_{C_i}} \sum_{y \in \mathbb{N}_{<x}} \textit{Granted}_{C_j}(t,y) + \\
& \min[\sum_{C_j \in \textit{LowerThan}_{C_i}} \sum_{y \in \mathbb{N}_{<x}} \textit{Granted}_{C_j}(t,y), \\
& \sum_{y \in \mathbb{N}_{<x}} \textit{Requested}_{C_i}(t,y) \cdot (LAT - 1) \;]
\end{aligned} \tag{7.43}$$

It is beyond the scope of this thesis to describe the use of this additional system property in co-runner-sensitive WCET analyses. The necessary changes with respect to the original co-runner-sensitive analyses (only relying on a work-conserving bus arbitration), however, are very similar to those for bounding the number of blocked cycles more precisely under Round-Robin arbitration (cf. the end of Section 7.5).

Due to time and space constraints, this thesis does not present any experimental results for systems with priority-based bus arbitration. It is the goal of this section to point out some of the particularities of priority-based bus arbitration protocols with respect to WCET analysis. Moreover, it is the goal of this section to give the reader an idea of the generality of our formal concepts (i.e. analyses for systems with priority-based bus arbitration can be derived in a similar way as the analyses for systems with Round-Robin arbitration). This concludes the brief discussion of priority-based bus arbitration protocols.

7.8. Consideration of Further Shared Resources

So far, we assumed a fixed latency LAT which every granted bus access takes until completion. This assumption holds in case the memory shared between the processor cores is a simple SRAM. It allows us to significantly decrease the degree of non-determinism during the modeling of the shared-bus interference (cf. the diagram in Figure 7.4 for a latency LAT of two clock cycles).

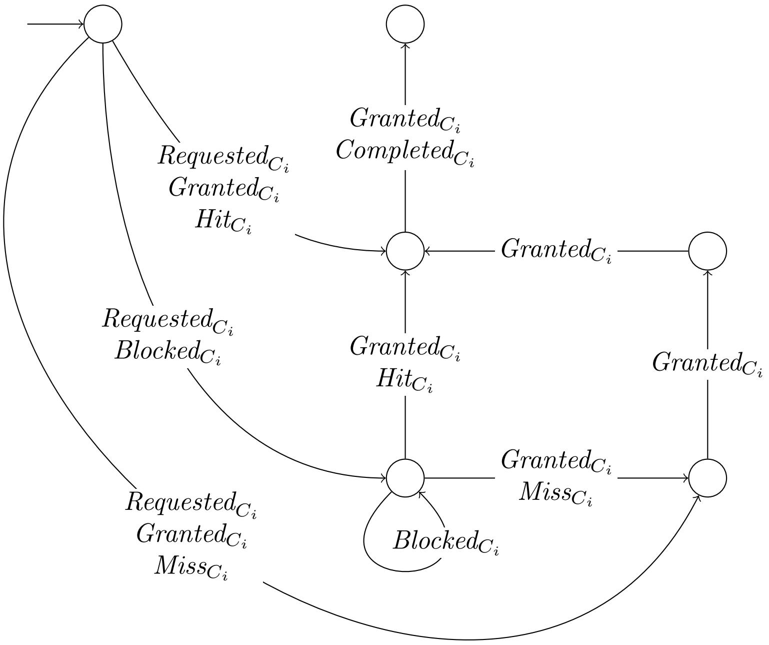


Figure 7.6.: Diagram representation of the non-determinism for modeling the shared-resource access behavior of a system with a shared-bus, a simple SRAM, and a shared cache between the bus and the SRAM. In case of a hit in the shared cache, the latency for a granted bus access shall be two clock cycles. In case of a cache miss, it shall be four clock cycles.

In a more realistic scenario, however, there is a *shared cache* between the shared bus and the shared memory. As a consequence, there is no longer one fixed latency which every granted bus access takes until completion. Instead, the actual latency of a particular granted bus access depends on whether the corresponding shared-cache access is a hit or a miss. Note that this is still safely covered by the very general modeling scheme presented in Figure 7.3 as it considers arbitrary latencies (≥ 1) per granted bus access.

In combination with a shared cache, there are typically two possible latencies which a granted bus access can take until completion—one for the case of a shared-cache hit and one for the case of a shared-cache miss.

$$\begin{aligned} &\exists LAT_{hit}, LAT_{miss} \in \mathbb{N}_{\geq 1} : \\ &\quad LAT_{hit} < LAT_{miss} \wedge \\ &\quad \forall t \in ExecRuns_{prog,C_i} : \forall acc \in getAccesses_{C_i}(t) \\ &\qquad \left|getGranted_{C_i}(t, acc)\right| \leq LAT_{miss} \wedge \\ &\qquad \left|getGranted_{C_i}(t, acc)\right| \notin \{LAT_{hit}, LAT_{miss}\} \Rightarrow getCompleted_{C_i}(t, acc) = \emptyset \end{aligned} \tag{7.44}$$

Consequently, we can create a non-determinism diagram that only features those two latencies. Figure 7.6 presents such a diagram for a cache-hit latency of two cycles and a cache-miss latency of four cycles. It safely models all possible cases for a concrete system with these latencies as it pessimistically assumes that every access might miss or hit the shared cache. These relatively short example latencies have primarily been chosen to keep the diagram small.

For such a system with two different bus-access latencies, there is no longer a single bus-access latency LAT. Many of the presented system properties bounding the shared-bus interference (e.g. P_{rr}, $P_{rr'}$, and $P_{AccGranu_1}$), however, argue about the latency LAT. For a system additionally containing a shared cache, corresponding system properties hold—pessimistically arguing about the cache-miss latency LAT_{miss} instead of LAT. Thus, we can safely reuse the presented analyses bounding the shared-bus interference for such a system by replacing LAT by LAT_{miss}.

While it is in principle sound to pessimistically assume that every access might miss or hit the shared cache, it results in a significant amount of overestimation with respect to the actual WCET [Yan and Zhang, 2008]. The precision can be improved by incorporating system properties that bound the amount of shared-cache interference that a processor core experiences. Existing approaches to WCET analysis for systems with shared caches [Yan and Zhang, 2008; Zhang and Yan, 2009; Li et al., 2009; Nagar and Srikant, 2014] implicitly rely on such system properties. A discussion of these system properties, however, is beyond the scope of this thesis.

The non-deterministic modeling scheme presented in Figure 7.6 makes the resulting WCET analyses applicable to systems exhibiting timing anomalies. Most existing approaches to WCET analysis for systems with shared caches [Yan and Zhang, 2008; Zhang and Yan, 2009; Li et al., 2009; Nagar and Srikant, 2014], in contrast, rely on timing compositionality [Hahn et al., 2013] and, thus, are not directly applicable to systems exhibiting timing anomalies. Thus, a combination of the modeling scheme that we propose and the system properties that the existing approaches implicitly rely on leads to WCET analyses safely modeling a significantly wider range of hardware platforms than the existing approaches. The derivation, implementation, and evaluation of such WCET analyses for systems with shared caches, however, exceed the scope of this thesis. Nonetheless, the principles presented in this thesis provide a good starting point for future work.

Note that a similar approach can also be chosen for modeling the overall interference in case there are other sources of interference influencing the overall latency of a granted bus access. This could e.g. be a second-level bus in a bus hierarchy, a DRAM performing regular refreshes, or a combination of multiple sources of interference (like a shared bus in front of a shared cache in front of a DRAM). The conceptual process of coming up with such non-deterministic modeling schemes is roughly described by three steps.

1. All sources of interference behind the shared bus are safely modeled by the scheme presented in Figure 7.3 as it considers arbitrary latencies (≥ 1) per granted bus access.

2. Additional knowledge is used to only enumerate latencies that are actually possible on the concrete system. In the resulting non-deterministic diagram (cf. Figure 7.6), the edges are labeled with system events (e.g. the longer latency is only possible in combination with a cache-miss event).

3. We use system properties to bound the amount of latency-inducing system events. We might e.g. know that the access behavior of the concurrent cores can never lead to more than a given number of misses at the shared cache for the core under analysis. Lifted versions of these system properties are used for the detection of infeasible abstract traces.

7.9. Discussion

The main advantage of incorporating the shared-resource interference already during WCET analysis is that we can subsequently use standard schedulability analyses that are not aware of the shared resources at all. However, the consideration of the interference during WCET analysis can also be a disadvantage in terms of precision. Consider the example in Figure 7.7: Programs $prog_1$ and $prog_2$ are executed on core C_1 and program $prog_3$ is executed on core C_2 of a dual-core processor. The scheduling shall be non-preemptive. There shall be one tiny program location in $prog_3$ (red dot) that is only executed once during its relatively long program execution run and that is the hot-spot in terms of interference generation on core C_2. The programs on core C_1, in contrast, are relatively short-running. Each of the programs on core C_1 can be interfered by the hot-spot on core C_2 during a run of the concrete system, which significantly increases

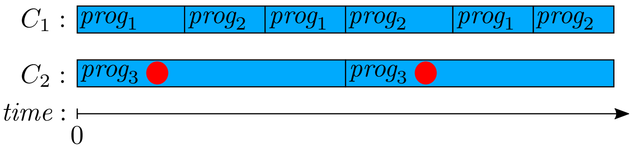


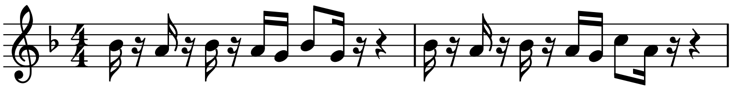
Figure 7.7.: Example for a cyclic scheduling in which at most one program on core C_1 can experience the maximal interference by core C_2 during a scheduling round. The red dot marks the hot-spot on core C_2 with respect to the interference it creates.

its execution time. However, if one of the programs is interfered by the hot-spot, the other is guaranteed to not be interfered by it during the same scheduling round of core C_1. If we account for the interference already during WCET analysis, each of the WCET bounds for programs $prog_1$ and $prog_2$ has to account for the interference generated by the hot-spot. Thus, a subsequent schedulability analysis which is not aware of the interference at all has to implicitly account twice for the interference generated by the hot-spot per scheduling round. This is clearly infeasible as the per-program worst cases exclude each other within the same scheduling round.

In order to overcome this potential pessimism, modern schedulability analyses are aware of shared-resource interference [Altmeyer et al., 2015]. They calculate the maximal amount of concurrent interference that a sequence of programs can experience. For the example in Figure 7.7, this could mean that the schedulability analysis finds out that three scheduling rounds on core C_1 can cumulatively only be interfered twice by the hot-spot of core C_2—instead of the six times implicitly assumed by a standard schedulability analysis relying on co-runner-sensitive WCET bounds.

Note, however, that essentially all existing schedulability analyses rely on timing compositionality [Hahn et al., 2013]. Thus, they are not directly applicable [Hahn et al., 2016a] to hardware platforms exhibiting timing anomalies [Lundqvist and Stenstrom, 1999]. In the next chapter, we bridge this gap by demonstrating the use of the techniques presented in this chapter for the calculation of bounds that can safely be used in analyses relying on timing compositionality. In this way, we enable the use of interference-aware schedulability analyses [Altmeyer et al., 2015] for the timing verification of systems exhibiting timing anomalies.

Chapter 8 Compositional Base Bounds



(El Plato Roto, Rafael Hernández)

Many WCET analyses and essentially all schedulability analyses rely on timing compositionality [Hahn et al., 2013]. Intuitively, these analyses start from a basic timing bound that assumes the absence of interference and subsequently add a fixed timing penalty per unit of interference. In a multi-core setting, the basic timing bound classically assumes the absence of shared-resource interference and, thus, is calculated using single-core WCET analysis techniques.

The determination of a safe timing penalty per unit of interference, however, remains an open problem [Hahn et al., 2016a] for hardware platforms exhibiting timing anomalies [Lundqvist and Stenstrom, 1999]. Due to timing anomalies, a clock cycle of being blocked at a shared bus may e.g. increase the overall execution time of the considered program by more than just one clock cycle. These timing anomalies have already been observed for surprisingly simple processor-core pipelines only featuring in-order execution [Hahn et al., 2016a]. They can be avoided by adapting the hardware design in a way that the processor-core pipeline is stalled whenever interference occurs [Hahn et al., 2016a]. Such changes to the hardware design, however, are expected to have a negative impact on the average-case performance of the system. Moreover, in most real-world scenarios, the design of the hardware platform must not be altered at all.

Consequently, it is so far unclear how to safely apply the verification approaches relying on timing compositionality to the majority of real-world hardware platforms exhibiting timing anomalies. We bridge this (widely ignored) gap by the calculation of *compositional base bounds*. Classically, the basic bounds for a timing-compositional analysis have been determined by single-core WCET analysis techniques which assume the absence of shared-resource interference. Compositional base bounds, in contrast, are calculated on top of safe techniques for modeling shared-resource interference (cf. Chapter 7). As a consequence, however, their calculation is computationally more complex than a corresponding WCET analysis which assume the absence of shared-resource interference.

In this chapter, we formalize the concept of compositional base bounds. Based on the principles of Chapter 7, we present two approaches to their calculation. Moreover, we discuss some of their applications.

8.1. The Concept of Compositional Base Bounds

A timing-compositional analysis typically tries to upper-bound the execution time of a program *prog* for a core C_i by starting from a basic bound and subsequently adding a fixed penalty per event that increases the execution time. In a more general scenario, we can as well upper-bound the number of occurrences of an arbitrary event E_{bnd}. Thus, timing compositionality corresponds to the special case of $E_{bnd} = \mathit{Cycle}$.

$$E_{bnd} \in \mathit{Events} \tag{8.1}$$

While upper-bounding event E_{bnd}, we add a fixed penalty *pen* per occurrence of event E. In general, we take into account different events E—each with a dedicated penalty from $\mathbb{R}_{\geq 0}$. This is reflected by the set *PenEv* of pairs of penalty and event.

$$\mathit{PenEv} \subseteq \mathbb{R}_{\geq 0} \times \mathit{Events} \tag{8.2}$$

We say that $R \in \mathbb{R}$ is a *compositional base bound* with respect to E_{bnd} and *PenEv* if—for any concrete program execution run—the number of occurrences of event E_{bnd} is *upper-bounded* ($\leq$) by R plus the numbers of occurrences of the events E from *PenEv* multiplied with their respective penalties.

$$\begin{aligned}
&\mathit{CompBaseBounds}_{prog,C_i}(E_{bnd}, \leq, \mathit{PenEv}) = \\
&\quad \{R \in \mathbb{R} \mid \forall t \in \mathit{ExecRuns}_{prog,C_i} : \\
&\qquad\qquad \mathit{numEvOccur}(prog, C_i, t, E_{bnd}) \leq \\
&\qquad\qquad\quad R + \sum_{(pen,E) \in PenEv} pen \cdot \mathit{numEvOccur}(prog, C_i, t, E)\}
\end{aligned} \tag{8.3}$$

Consider the example of $R \in \mathit{CompBaseBounds}_{prog,C_i}(\mathit{Cycle}, \leq, \{(pen_1, E_1)\})$. Its intuitive meaning is that every program execution run with at most x occurrences of event E_1 cannot have an execution time greater than $R + pen_1 \cdot x$. Thus, we can safely use R as base bound in a timing-compositional analysis that accounts for every occurrence of event E_1 by adding a penalty of pen_1 clock cycles.

Note, however, that—for particular choices of E_{bnd} and *PenEv*—the set of compositional base bounds may be empty. Consider, e.g., a multi-core processor with a shared bus, priority-based bus arbitration, and a program *prog* that suffers from starvation on core C_i. For this concrete scenario, the set $\mathit{CompBaseBounds}_{prog,C_i}(\mathit{Cycle}, \leq, \{(0.5, \mathit{Blocked}_{C_i})\})$ is empty, as there can be an unbounded amount of blocked cycles (cf. Section 7.7) and—as soon as the pipeline is converged during an access to the shared bus (cf. Section 9.2)—each of them contributes a full clock cycle to the execution time. Thus, there is no compositional base bound unless we consider a penalty of at least one clock cycle per cycle blocked at the shared bus. On the other hand, for any program which has a finite WCET, there is even a compositional base bound under a penalty of zero clock cycles per cycle blocked at the shared bus.

So far, we only defined compositional base bounds for upper-bounding the number of occurrences of event E_{bnd}. Analogously, we can define compositional base bounds such that the number of occurrences of event E_{bnd} is *lower-bounded* ($\geq$) by R plus the numbers of occurrences of the events E from *PenEv* multiplied with their respective penalties.

$$\begin{aligned} &\mathit{CompBaseBounds}_{prog,C_i}(E_{bnd}, \geq, \mathit{PenEv}) = \\ &\quad \{R \in \mathbb{R} \mid \forall t \in \mathit{ExecRuns}^{min\text{-}relev}_{prog,C_i,E} : \\ &\qquad\qquad \mathit{numEvOccur}(prog, C_i, t, E_{bnd}) \geq \\ &\qquad\qquad R + \sum_{(pen,E)\in PenEv} pen \cdot \mathit{numEvOccur}(prog, C_i, t, E)\} \end{aligned} \tag{8.4}$$

An upper-bounding compositional base bound is more precise than another one if it is smaller. Analogously, a lower-bounding compositional base bound is more precise than another one if it is greater.

This concludes the formal definition of compositional base bounds with respect to the concrete traces. In order to efficiently calculate compositional base bounds, we resort to the higher levels of approximation described in the previous chapters. In the following sections, we present two calculation procedures.

8.2. Calculation by Subtraction of Edge Weights

In this section, we present a calculation procedure for compositional base bounds that operates on a graph G^B as used for the calculation of event bounds using implicit path enumeration (cf. Section 6.5). The calculation procedure also relies on implicit path enumeration.

According to the above definitions, compositional base bounds have to bound the following difference—either from above (cf. equation (8.3)) or from below (cf. equation (8.4)).

$$\mathit{numEvOccur}(prog, C_i, t, E_{bnd}) - \sum_{(pen,E)\in PenEv} pen \cdot \mathit{numEvOccur}(prog, C_i, t, E)$$

The procedure calculates the corresponding difference per graph edge. Subsequently, these differences are used as edge weights in an implicit path enumeration. For the calculation of upper-bounding compositional base bounds, it uses upper-bounding edge weights for event E_{bnd} and lower-bounding edge weights for the events E from *PenEv*.

$$\begin{aligned} \mathit{Maximum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} = &\max_{(tt,*,*)\in \widehat{LessImplicit}^B} \sum_{edg\in Edges^B} tt(edg) \cdot \\ &\qquad [\widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(edg) \\ &\qquad - \sum_{(pen,E)\in PenEv} pen \cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(edg)] \end{aligned} \tag{8.5}$$

Analogously, for the calculation of lower-bounding compositional base bounds, it uses lower-bounding edge weights for event E_{bnd} and upper-bounding edge weights for the events E from *PenEv*.

$$\begin{aligned} \mathit{Minimum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} = &\min_{(tt,*,*)\in \widehat{LessImplicit}^B} \sum_{edg\in Edges^B} tt(edg) \cdot \\ &\qquad [\widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(edg) \\ &\qquad - \sum_{(pen,E)\in PenEv} pen \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(edg)] \end{aligned} \tag{8.6}$$

Whenever the procedure results in a defined value, this value is guaranteed to be a compositional base bound. This is reflected by the following two statements. For a proof sketch for these statements, we refer to page 313.

$$\begin{aligned}&\mathit{Maximum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} \in \mathbb{R}\\ \Rightarrow &\mathit{Maximum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} \in \mathit{CompBaseBounds}_{prog,C_i}(E_{bnd}, \leq, \mathit{PenEv})\end{aligned} \tag{8.7}$$

$$\begin{aligned}&\mathit{Minimum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} \in \mathbb{R}\\ \Rightarrow &\mathit{Minimum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} \in \mathit{CompBaseBounds}_{prog,C_i}(E_{bnd}, \geq, \mathit{PenEv})\end{aligned} \tag{8.8}$$

The implicit path enumeration used in this calculation procedure is typically realized by integer linear programming (ILP). A performance-optimized implementation of the procedure would statically evaluate the difference per graph edge during the construction of the integer linear program. It is, however, as well possible to explicitly perform the subtraction during the ILP-solving step. To this end, one introduces an additional integer variable $var_{(pen,E)}$ per $(pen, E) \in \mathit{PenEv}$. For the calculation of upper-bounding compositional base bounds, additional constraints encode that the value of $var_{(pen,E)}$ must never be smaller than the sum over the corresponding lower-bounding edge weights along the implicit path.

$$\forall (pen, E) \in \mathit{PenEv}: \sum_{edg \in Edges^B} tt(edg) \cdot \widehat{wEvent^{LB}_{prog,C_i,E}}(edg) \leq var_{(pen,E)} \tag{8.9}$$

The actual calculation of an upper-bounding compositional base bounds is then performed as follows.

$$\begin{aligned}\max_{\substack{(tt,*,*) \in \widehat{LessImplicit^B}\\ \forall (pen,E) \in PenEv: var_{(pen,E)} \in \mathbb{Z}}} \Bigg[&\sum_{edg \in Edges^B} tt(edg) \cdot \widehat{wEvent^{UB}_{prog,C_i,E_{bnd}}}(edg)\\ &- \sum_{(pen,E) \in PenEv} pen \cdot var_{(pen,E)} \Bigg]\end{aligned} \tag{8.10}$$

A corresponding implementation of the calculation of lower-bounding compositional base bounds is obtained by switching the bound directions in equations (8.9) and (8.10), replacing the $\leq$ in equation (8.9) by $\geq$, and replacing the maximum in equation (8.10) by a minimum.

In a recent research paper [Hahn et al., 2016a], we have presented this style of implementation for the calculation of upper-bounding compositional base bounds. On page 314, we formally argue that it is indeed an implementation of the calculation procedure presented in this section (cf. equations (8.5) and (8.6)) as it is guaranteed to provide the same results.

A performance comparison between this alternative style of implementation and an implementation that already evaluates the subtractions while formulating the implicit path enumeration is beyond the scope of this thesis.

The precision of the compositional base bound calculated by the presented procedure depends on the graph representation on which the calculation operates. In particular, the choice of the edge-weight-sensitivity (cf. Section 6.4.4) of the graph with respect to the events in *PenEv* is important. A case study on the actual impact of the edge-weight-sensitivity, however, is beyond the scope of this thesis. In our first experiments with the calculation of compositional base bounds [Hahn et al., 2016a], we aimed for the best precision possible by relying on a full edge-weight-sensitivity for the events in *PenEv*. A full edge-weight-sensitivity, however, significantly increases the graph size and, thus, also the runtime of the graph construction (cf. Section 6.4.4) and the implicit path enumeration (cf. Section 6.5). Thus, the presented calculation procedure might suffer from scalability issues when aiming for the best precision possible. In order to improve the scalability, in the next section, we present a calculation procedure that performs the subtraction already during the initial graph construction. In this way, a good precision of the resulting compositional base bound does not depend on the edge-weight-sensitivity for the events in *PenEv*.

8.3. Calculation by Subtraction during Graph Construction

In this section, we demonstrate the calculation of compositional base bounds by performing a subtraction already during the initial graph construction from the results of the micro-architectural analysis. To this end, we slightly extend Algorithm 6.3, which is responsible for the creation of the edges and edge weights during the graph construction presented in Section 6.4.4.

The code snippet in Algorithm 8.1 shows the extended part of Algorithm 6.3. The added lines are highlighted. The extension of Algorithm 8.1 adds edge weights for the dummy event $\mathit{Event}_{prog,C_i,Cmp}$. In contrast to the edge weights for the actual events (cf. equations (5.58) and (5.62)), the added edge weights can be negative and/or non-integer. The shorthand to access these weights along the positions of a path shall exist in the same way as for the edge weights of the actual events (cf. equations (5.60) and (5.61)).

$$\widehat{\mathit{wEvent}^{UB}_{prog,C_i,Cmp}}, \widehat{\mathit{wEvent}^{LB}_{prog,C_i,Cmp}} : \mathit{Edges} \to \mathbb{R} \tag{8.11}$$

The values assigned to these new edge weights bound the difference between the number of occurrences of event E_{bnd} and the sum over the numbers of occurrences of the events E from *PenEv* multiplied with their respective penalties (cf. Algorithm 8.1).

For the actual calculation of compositional base bounds, we reuse the machinery for the calculation of event bounds via implicit path enumeration from Section 6.5 (cf. equations (6.173) and (6.174)). We use this machinery for the calculation of bounds for the dummy event *Cmp*. In case this calculation results in a defined value, this value is a compositional base bound. This is expressed by the following two statements. A formal proof of these statement is omitted due to time and space constraints.

$$\begin{aligned} &\widehat{\mathit{Maximum}^{B,impli}_{prog,C_i,Cmp}} \in \mathbb{R} \\ \Rightarrow &\widehat{\mathit{Maximum}^{B,impli}_{prog,C_i,Cmp}} \in \mathit{CompBaseBounds}_{prog,C_i}(E_{bnd}, \leq, \mathit{PenEv}) \end{aligned} \tag{8.12}$$

$$\begin{aligned} &\widehat{\mathit{Minimum}^{B,impli}_{prog,C_i,Cmp}} \in \mathbb{R} \\ \Rightarrow &\widehat{\mathit{Minimum}^{B,impli}_{prog,C_i,Cmp}} \in \mathit{CompBaseBounds}_{prog,C_i}(E_{bnd}, \geq, \mathit{PenEv}) \end{aligned} \tag{8.13}$$

This concludes the presentation of calculation procedures for compositional base bounds. In the following sections, we discuss applications of compositional base bounds.

8.4. Using Compositional Base Bounds in Existing Schedulability Analyses

Compositional base bounds effectively bridge the gap between existing schedulability analyses relying on timing compositionality and modern hardware platforms exhibiting timing anomalies.

Note, however, that schedulability analyses aware of shared-resource interference typically start from the optimistic assumption that there is no interference [Schliecker et al., 2008; Altmeyer et al., 2015]. Subsequently, they calculate values on arrival curves for the concurrent cores and add more

Algorithm 8.1 : Code snippet showing the extended part of Algorithm 6.3, which enables the calculation of compositional base bounds. It adds edge weights for the dummy event *Cmp*. The added lines are highlighted.

```
...
for $E \in Events$ do
    $\widehat{wE}^{UB}((nd_1, targetNd)) \longleftarrow \max_{\widehat{p} \in set} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wE}^{UB}(\widehat{p}, x)$;
    $\widehat{wE}^{LB}((nd_1, targetNd)) \longleftarrow \min_{\widehat{p} \in set} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wE}^{LB}(\widehat{p}, x)$;
$\widehat{wEvent}^{UB}_{prog,C_i,Cmp}((nd_1, targetNd)) \longleftarrow \max_{\widehat{p} \in set} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} [\widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(\widehat{p}, x)$
        $- \sum_{(pen,E) \in PenEv} pen \cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(\widehat{p}, x)]$;
$\widehat{wEvent}^{LB}_{prog,C_i,Cmp}((nd_1, targetNd)) \longleftarrow \min_{\widehat{p} \in set} \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} [\widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(\widehat{p}, x)$
        $- \sum_{(pen,E) \in PenEv} pen \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(\widehat{p}, x)]$;
if $targetNd \neq nd_2$ then
    $Edges^{constr} \longleftarrow Edges^{constr} \cup \{(targetNd, nd_2)\}$;
    for $E \in Events$ do
        $\widehat{wE}^{UB}((targetNd, nd_2)) \longleftarrow 0$;
        $\widehat{wE}^{LB}((targetNd, nd_2)) \longleftarrow 0$;
    $\widehat{wEvent}^{UB}_{prog,C_i,Cmp}((targetNd, nd_2)) \longleftarrow 0$;
    $\widehat{wEvent}^{LB}_{prog,C_i,Cmp}((targetNd, nd_2)) \longleftarrow 0$;
...
```

and more interference. As discussed in Section 7.5, such an approach does not necessarily lead to a fixed point which is sound with respect to the concrete traces (i.e. assumption (A.140) does in general not necessarily hold for the abstract models on which the compositional base bounds are calculated). Thus, the soundness of the compositional base bounds does not necessarily imply the soundness of the overall timing verification for these approaches.

In order to still enable the sound use of compositional base bounds in combination with these schedulability analyses, we propose a special flavor of compositional base bounds. Analogously to the adapted version of our co-runner-sensitive analysis starting from an optimistic initialization (cf. equations (7.30) and (7.32)), we only argue about the events E from $PenEv$ that happen at a non-tail position. We only present the upper-bounding version of this special flavor of compositional base bounds.

$$\begin{aligned} &CompBaseBounds'_{prog,C_i}(E_{bnd}, \leq, PenEv) = \\ &\quad \{R \in \mathbb{R} \mid \forall t \in ExecRuns_{prog,C_i} : \\ &\qquad numEvOccur(prog, C_i, t, E_{bnd}) \leq \\ &\qquad\quad R + \sum_{(pen,E) \in PenEv} pen \cdot numEvOccur'(prog, C_i, t, E)\} \end{aligned} \tag{8.14}$$

$$numEvOccur'(prog, C_i, t, E) = \left|\{x \in \mathbb{N}_{<len(t)-1} \mid Event_{prog,C_i,E}(t, x)\}\right| \tag{8.15}$$

We slightly adapt the presented calculation procedures in order to result in this flavor of compositional base bounds. The calculation procedure of Section 8.2 is adapted as follows. For a safe result, every node of the underlying graph representation has to be marked as an end node (i.e. we are interested in all path prefixes of the original graph).

$$\begin{aligned} Maximum^{\widehat{B,impli\prime}}_{prog,C_i,E_{bnd},PenEv} = \max_{(tt,*,isEnd)\in Less\widehat{Implicit}^B} \sum_{edg\in Edges^B} & [tt(edg)\cdot \widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(edg) \\ & - tt(edg)\cdot \sum_{(pen,E)\in PenEv} pen\cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(edg) \\ & + isEnd(edg)\cdot \sum_{(pen,E)\in PenEv} pen\cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(edg)] \end{aligned} \tag{8.16}$$

The calculation procedure of Section 8.3, in contrast, is adapted as follows. The graph construction stays as described by Algorithm 8.1. This time, however, we use the edge weight $\widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}$ instead of $\widehat{wEvent}^{UB}_{prog,C_i,Cmp}$ for the last edge on the path. This variant also relies on every node of the underlying graph to be marked as an end node.

$$\begin{aligned} Maximum^{\widehat{B,impli\prime}}_{prog,C_i,Cmp,E_{bnd}} = \max_{(tt,*,isEnd)\in Less\widehat{Implicit}^B} \sum_{edg\in Edges^B} & [tt(edg)\cdot \widehat{wEvent}^{UB}_{prog,C_i,Cmp}(edg) \\ & - isEnd(edg)\cdot \widehat{wEvent}^{UB}_{prog,C_i,Cmp}(edg) \\ & + isEnd(edg)\cdot \widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(edg)] \end{aligned} \tag{8.17}$$

A formal soundness proof of these adapted calculation procedures is beyond the scope of this thesis. In the same way, it exceeds the scope of this thesis to formally argue about the soundness of using the special flavor of compositional base bounds in schedulability analyses starting from an optimistic initialization and calculating values on arrival curves.

When we began to work out the idea of compositional base bounds, we only planned to use penalties which coincide with the direct effect that a unit of interference has on the timing (e.g. one cycle per cycle blocked at the shared bus or LAT cycles per concurrently granted access). Soon after, Jan Reineke pointed out that we can as well use other penalties—as long as they result in a defined compositional base bound. In general, it depends on the actual amount of concurrent interference which penalties—in combination with the corresponding compositional base bound—will lead to the best precision when used during an interference-aware schedulability analysis. This is demonstrated in the example of Figure 8.1. R_1 (R_2) is a compositional base bound that upper-bounds the number of occurrences of event E_{bnd} by assuming a penalty of pen_1 (pen_2) per occurrence of event E. In case there are less than x occurrences of event E, R_1 provides a more precise upper bound in combination with penalty pen_1. In case there are more than x occurrences of event E, R_2 provides a more precise upper bound in combination with penalty pen_2.

Future schedulability analyses might exploit this by supporting multiple base bounds—each one with a dedicated set of penalties. Thus, such schedulability analyses would always calculate multiple time bounds and subsequently use the most precise one. A detailed discussion of such schedulability analyses, however, is beyond the scope of this thesis.

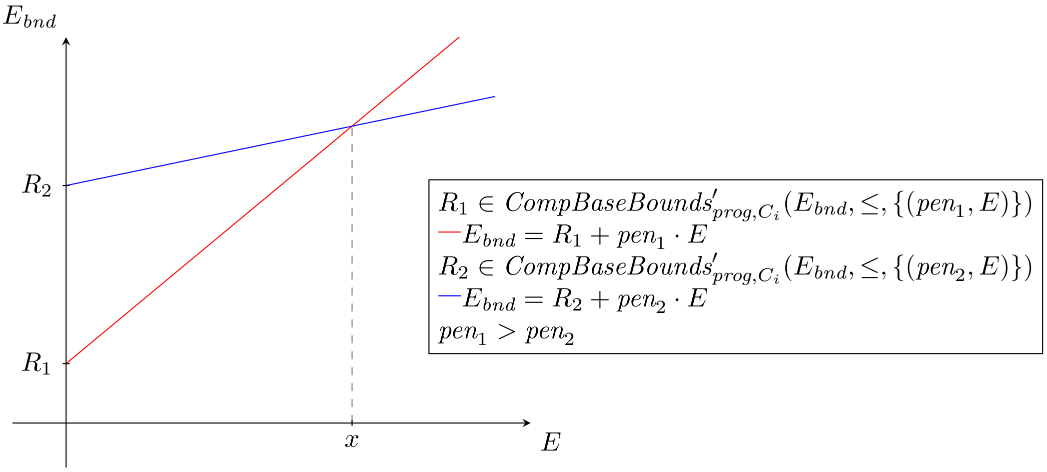


Figure 8.1.: Example in which it depends on the actual number of occurrences of event E which penalty and corresponding compositional base bound lead to a more precise upper bound on the number of occurrences of event E_{bnd}.

8.5. Sketch: Replacing Constraints by Compositionality for Path Analysis

During implicit path enumeration, lifted versions of system properties are typically used to improve the precision of the calculated WCET bound [Li and Malik, 1995; Engblom and Ermedahl, 2000; Stein, 2010; Cullmann, 2013; Raymond, 2014; Nagar and Srikant, 2015; Blaß et al., 2017]. The soundness of this approach is a direct consequence of the lifted versions fulfilling the soundness criteria for lifting properties up the hierarchy of abstract models presented in Chapter 5.

The linear constraints encoding the lifted properties, however, typically increase the runtime of implicit path enumeration. In this section, we sketch how to replace linear constraints by compositionality. We expect the sketched approach to provide a trade-off between the precision of incorporating a set of linear constraints during implicit path enumeration and the efficiency of not incorporating it. An experimental evaluation of the sketched approach, however, is beyond the scope of this thesis.

We sketch our approach for the well-known property of cache persistence [Cullmann, 2013]. Intuitively, it states that—per execution of a certain scope of the program under analysis—there may at most be one cache miss for a memory address that has been classified as persistent in the scope. This can be expressed by linear constraints in the following way for a set of persistent addresses and corresponding scopes.

$$\begin{aligned} &\forall (addr, scope) \in AddrPersistentInScope: \\ &\quad \sum_{edg \in Edges^B} tt(edg) \cdot w\widehat{Event^{LB}_{prog,C_i,Miss_{addr,scope}}}(edg) \\ &\quad \leq \sum_{edg \in Edges^B} tt(edg) \cdot w\widehat{Event^{UB}_{prog,C_i,Enter_{scope}}}(edg) \end{aligned} \tag{8.18}$$

In order to improve the precision, these constraints are added to an implicit path enumeration calculating a WCET bound.

$$\max_{(tt,*,*) \in \widehat{LessImplicit}^B} \sum_{edg \in Edges^B} tt(edg) \cdot w\widehat{Event^{UB}_{prog,C_i,Cycle}}(edg) \tag{8.19}$$

We learned from our industry partner[1] that the additional cache persistence constraints often lead to a dramatic increase of the runtime of implicit path enumeration. Thus, for real-world scenarios, the determination of persistent cache accesses and the generation of the corresponding persistence constraints are often omitted in order to keep the complexity of implicit path enumeration at a manageable level.

We aim at avoiding the additional complexity of cache persistence constraints and, yet, profiting—to some degree—from cache persistence information. To this end, we propose to use the graph construction presented in Section 8.3 as it would also be used for the calculation of a compositional base bound from the following set.

$$\begin{aligned} &\mathit{CompBaseBounds}_{\mathit{prog},C_i}(\mathit{Cycle}, \leq, \{(\mathit{pen}_{\mathit{miss}}, \mathit{Miss}_{\mathit{addr},\mathit{scope}}) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \mid (\mathit{addr}, \mathit{scope}) \in \mathit{AddrPersistentInScope}\}) \end{aligned}$$

Based on the resulting graph, we propose the following implicit path enumeration for the calculation of a WCET bound.

$$\begin{aligned} \max_{(tt,*,*)\in \widehat{\mathit{LessImplicit}}^B} \sum_{\mathit{edg}\in \mathit{Edges}^B} \mathit{tt}(\mathit{edg}) \cdot [&\widehat{\mathit{wEvent}}^{UB}_{\mathit{prog},C_i,\mathit{Cmp}}(\mathit{edg}) + \\ &\sum_{(\mathit{addr},\mathit{scope})\in \mathit{AddrPersistentInScope}} \\ &\mathit{pen}_{\mathit{miss}} \cdot \widehat{\mathit{wEvent}}^{UB}_{\mathit{prog},C_i,\mathit{Enter}_{\mathit{scope}}}(\mathit{edg})] \end{aligned} \tag{8.20}$$

The sketched approach adds a cache miss penalty per time the scope of a persistent cache access is entered. A resulting WCET bound is sound with respect to the concrete traces. Intuitively, the soundness is a consequence of the soundness of the compositional base bound that is a part of the calculated WCET bound. A formal soundness proof, however, exceeds the scope of this thesis.

Note that the value of the square bracket can be evaluated statically while generating the formulation of the implicit path enumeration. Thus, we expect the proposed approach to have a similar runtime as the WCET bound calculation which does not take into account any cache persistence information (i.e. following equation (8.19), but not adding the persistence constraints of equation (8.18)).

We expect the proposed approach to often result in a WCET bound that is more precise than the WCET bound calculated without taking into account any cache persistence information. In some cases, however, it might result in a less precise WCET bound. If it is strictly required to not be less precise than a WCET analysis not taking into account any cache persistence information, we recommend to calculate both WCET bounds (i.e. following equation (8.19) and following equation (8.20)) and to subsequently choose the more precise one. An optimized implementation of this might calculate the minimum of both WCET bounds during a single implicit path enumeration.

Sebastian Hahn conducted first experiments in order to evaluate the computational efficiency and precision of the sketched approach. These experiments reveal that the classical approach of relying on persistence constraints (i.e. following equations (8.19) and (8.18)) only leads to a significant increase of the computational complexity compared to not taking into account any cache persistence information (i.e. following equation (8.19), but not adding the persistence constraints of equation (8.18)) for the most complex hardware platform that we consider (i.e. out-of-order execution with unblocked stores) in combination with the most complex benchmarks that we consider. Thus, we expect that the sketched approach is more relevant for real-world analysis scenarios with hardware platforms and benchmarks that are significantly more complex than what we consider. Moreover, these first experiments indicate that, in order to achieve a reasonable precision, it will be necessary to consider an individual penalty per pair of memory

[1] https://www.absint.com

address and persistence scope. Intuitively, this is caused by the maximal time difference between a cache hit and a persistent cache miss of the same memory block differing significantly for different pairs of memory address and persistence scope due to different constellations in the processor pipeline. A more detailed discussion of these first experiments and their preliminary results, however, exceeds the scope of this thesis.

Part III.

Implementation of WCET Analyses

Chapter 9 Evaluation of the Co-Runner-Insensitive WCET Analysis

Es gibt viel zu verlieren, du kannst nur gewinnen.
Genug ist zu wenig, oder es wird so wie es war.
Stillstand ist der Tod, geh voran, bleibt alles anders.
Der erste Stein fehlt in der Mauer.
Der Durchbruch ist nah.

(Bleibt alles anders, Herbert Grönemeyer, 1998)

In this chapter, we evaluate the presented co-runner-insensitive WCET analysis for multi-core processors with shared buses (cf. Section 7.4). We demonstrate that a naive implementation of the analysis leads to a significant increase in analysis runtime compared to an analysis assuming the absence of shared-bus interference (on average around seven times as long for a relatively simple processor core). In order to reduce this overhead, we present two key implementation tricks.

We evaluate the different implementations of our WCET analysis for a *quad-core processor* with a *shared bus* and a *Round-Robin* bus arbitration policy. There is *no shared cache* on the considered hardware platform. The shared bus connects the processor cores to an *SRAM memory*. We assume a fixed latency of 13 clock cycles per granted access to the shared bus (ten cycles for the first word of a four-word cache line, one cycle for every following word). Figure 9.1 sketches the schematic design of the considered hardware platform.

In our experiments, we consider different processor core configurations. We consider processor cores with five-stage in-order pipelines as well as cores with out-of-order pipelines (Tomasulo, four functional units) similar to those described in [Hennessy and Patterson, 2011]. Furthermore,

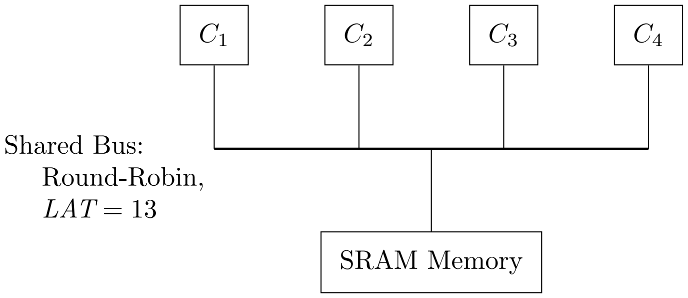


Figure 9.1.: Schematic system design of the considered hardware platform: a quad-core processor with a shared bus, round-robin bus arbitration, and an SRAM memory. We assume a fixed latency of 13 clock cycles per granted bus access.

	In-Order Pipeline	Out-of-Order Pipeline
Local Instruction Scratchpad	$Conf_{is}^{io}$	$Conf_{is}^{ooo}$
Local Instruction Cache	$Conf_{ic}^{io}$	$Conf_{ic}^{ooo}$

Table 9.1.: The processor core configurations that we consider in our experiments.

we also distinguish two scenarios with respect to the local instruction memories of the processor cores. First, we assume a local instruction scratchpad per core that is large enough to be statically initialized with all programs executed on the core. Secondly, we consider a local instruction cache of size 1KiB that is connected to the shared bus. Table 9.1 lists the four resulting processor core configurations. All configurations assume a local data cache of size 1KiB that is connected to the shared bus.

In our experiments, we calculate WCET bounds for a subset of the programs of the TACLeBench suite [Falk et al., 2016]. Note that we had to exclude some of the benchmarks of the TACLeBench suite from the experiments because the prototype implementation of our analysis tool LLVMTA [Jacobs et al., 2015; Hahn et al., 2016a] does not yet support them (e.g. due to the use of recursion in these benchmarks). Moreover, we calculate WCET bounds for programs generated from models developed in the SCADE Suite®[1]—including the examples delivered with the SCADE Suite. Table 9.2 lists the 47 benchmarks that we consider in our experiments. In terms of size, the benchmarks range from around 100 to around 13000 lines of code. We assume that these programs are *scheduled non-preemptively* as our analysis does not take into account any preemption costs [Altmeyer, 2013].

The evaluated calculation of co-runner-insensitive WCET bounds is performed on graphs that are fully node-sensitivity at basic block boundaries, node-insensitive inside of basic blocks, and edge-weight-insensitive (cf. Section 6.4.4).

We conduct all experiments on a quad-core Intel® Core™ i7 processor clocked at 2.4 GHz and provided 16 GiB of main memory.

9.1. A Naive Implementation

A naive implementation of the micro-architectural analysis explores all sequences of abstract states that are possible for an access to the shared bus according to our non-deterministic overapproximation of the shared-bus interference (cf. Chapter 7). This is demonstrated for a single bus access in Figure 9.2. For simplicity, the example assumes a dual-core processor with a granted-access latency of three clock cycles and Round-Robin bus arbitration. Thus, every access to the shared bus can be blocked (**B**) for up to three cycles (cf. equation (7.13)) before it is granted (**G**) for three cycles.

We experimentally evaluate the analysis runtime and memory consumption of the presented naive implementation during the calculation of co-runner-insensitive WCET bounds. The programs under analysis (cf. Table 9.2) are assumed to be executed on the quad-core processor sketched in Figure 9.1 with the processor core configuration $Conf_{is}^{io}$ from Table 9.1.

The diagram in Figure 9.3 shows the analysis runtime per benchmark normalized to the corresponding runtime of a single-core analysis (i.e. assuming the absence of shared-bus interference). The analysis runtime is between 2.21 times (for benchmark `binarysearch`) and 12.65 times (for benchmark `flight_control`) as high as for a single-core analysis. On average, the analysis takes 6.92 times as long as a single-core analysis. The average factor of 6.92 has been calculated as the geometric mean of the per-benchmark factors. The overall experiment (i.e. the analysis of all considered programs) has a runtime of 17 minutes and 19.95 seconds (compared to one minute and 57 seconds for a corresponding experiment assuming the absence of shared-bus interference).

[1] http://www.esterel-technologies.com/products/scade-suite

Benchmark	Suite	Lines of Code
cruise_control	scadetests	1225
digital_stopwatch	scadetests	874
es_lift	scadetests	1120
flight_control	scadetests	3739
pilot	scadetests	1587
roboDog	scadetests	1792
trolleybus	scadetests	3920
lift	taclebench/app	565
powerwindow	taclebench/app	2277
binarysearch	taclebench/kernel	156
bsort	taclebench/kernel	131
complex_updates	taclebench/kernel	135
countnegative	taclebench/kernel	140
fft	taclebench/kernel	651
filterbank	taclebench/kernel	170
fir2dim	taclebench/kernel	198
iir	taclebench/kernel	163
insertsort	taclebench/kernel	139
jfdctint	taclebench/kernel	318
lms	taclebench/kernel	180
ludcmp	taclebench/kernel	179
matrix1	taclebench/kernel	169
md5	taclebench/kernel	625
minver	taclebench/kernel	266
pm	taclebench/kernel	2061
prime	taclebench/kernel	140
sha	taclebench/kernel	2349
st	taclebench/kernel	226
adpcm_dec	taclebench/sequential	712
adpcm_enc	taclebench/sequential	751
audiobeam	taclebench/sequential	7010
cjpeg_transupp	taclebench/sequential	1469
cjpeg_wrbmp	taclebench/sequential	1775
dijkstra	taclebench/sequential	311
epic	taclebench/sequential	1207
g723_enc	taclebench/sequential	878
gsm_dec	taclebench/sequential	1734
gsm_encode	taclebench/sequential	3168
h264_dec	taclebench/sequential	1433
huff_dec	taclebench/sequential	383
mpeg2	taclebench/sequential	13212
ndes	taclebench/sequential	374
petrinet	taclebench/sequential	979
rijndael_dec	taclebench/sequential	3281
rijndael_enc	taclebench/sequential	3117
statemate	taclebench/sequential	1278
susan	taclebench/sequential	10237

Table 9.2.: The 47 benchmarks that we consider in our experiments. They are generated from models developed in the SCADE Suite® or taken from the TACLeBench suite.

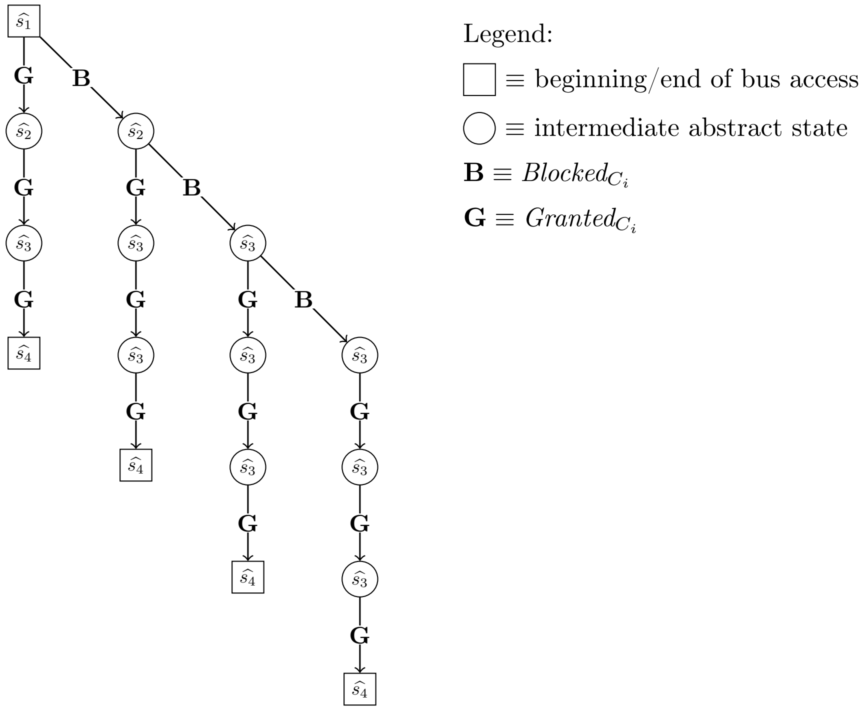

$\widehat{s}_1$
G
B
$\widehat{s}_2$
$\widehat{s}_3$
$\widehat{s}_4$
Legend:
□ ≡ beginning/end of bus access
○ ≡ intermediate abstract state
B ≡ *Blocked*$_{C_i}$
G ≡ *Granted*$_{C_i}$

Figure 9.2.: In a naive implementation, for every bus access, the micro-architectural analysis explores all sequences of abstract states that are possible according to the non-deterministic overapproximation of the shared-bus interference. For simplicity, this example assumes a dual-core processor with a granted-access latency of three clock cycles and Round-Robin bus arbitration. Thus, every access to the shared bus can be blocked (**B**) for up to three cycles (cf. equation (7.13)) before it is granted (**G**) for three cycles.

Analogously, the diagram in Figure 9.4 shows the analysis memory consumption per benchmark normalized to the corresponding memory consumption of a single-core analysis (i.e. assuming the absence of shared-bus interference). The memory consumption increases by up to 15 percent (for benchmark `susan`) compared to a single-core analysis. For benchmark `filterbank`, it even decreases by three percent. On average, the analysis consumes four percent more memory than an analysis assuming the absence of shared-bus interference.

While the naive implementation only consumes slightly (up to 15 percent) more memory than a single-core analysis, its increase in analysis runtime compared to a single-core analysis is significant (up to 12.65 times as long, on average 6.92 times as long). We expect the average increase in analysis runtime of the naive implementation to be even higher for more complex processor core configurations. An experimental evaluation of the naive implementation for more complex processor core configurations, however, is omitted due to space and time constraints.

Note that, in an earlier publication [Jacobs et al., 2015], we have reported a significantly higher increase in analysis runtime (on average 38.84 times as long) and also a significant increase in memory consumption (on average 3.62 times as much) for the naive implementation (quad-core, $Conf_{is}^{io}$) compared to a single-core analysis. The improved factors in this thesis are the result of many engineering improvements in the analysis framework that our implementation prototype uses. A detailed discussion of these engineering improvements, however, is beyond the scope of this thesis.

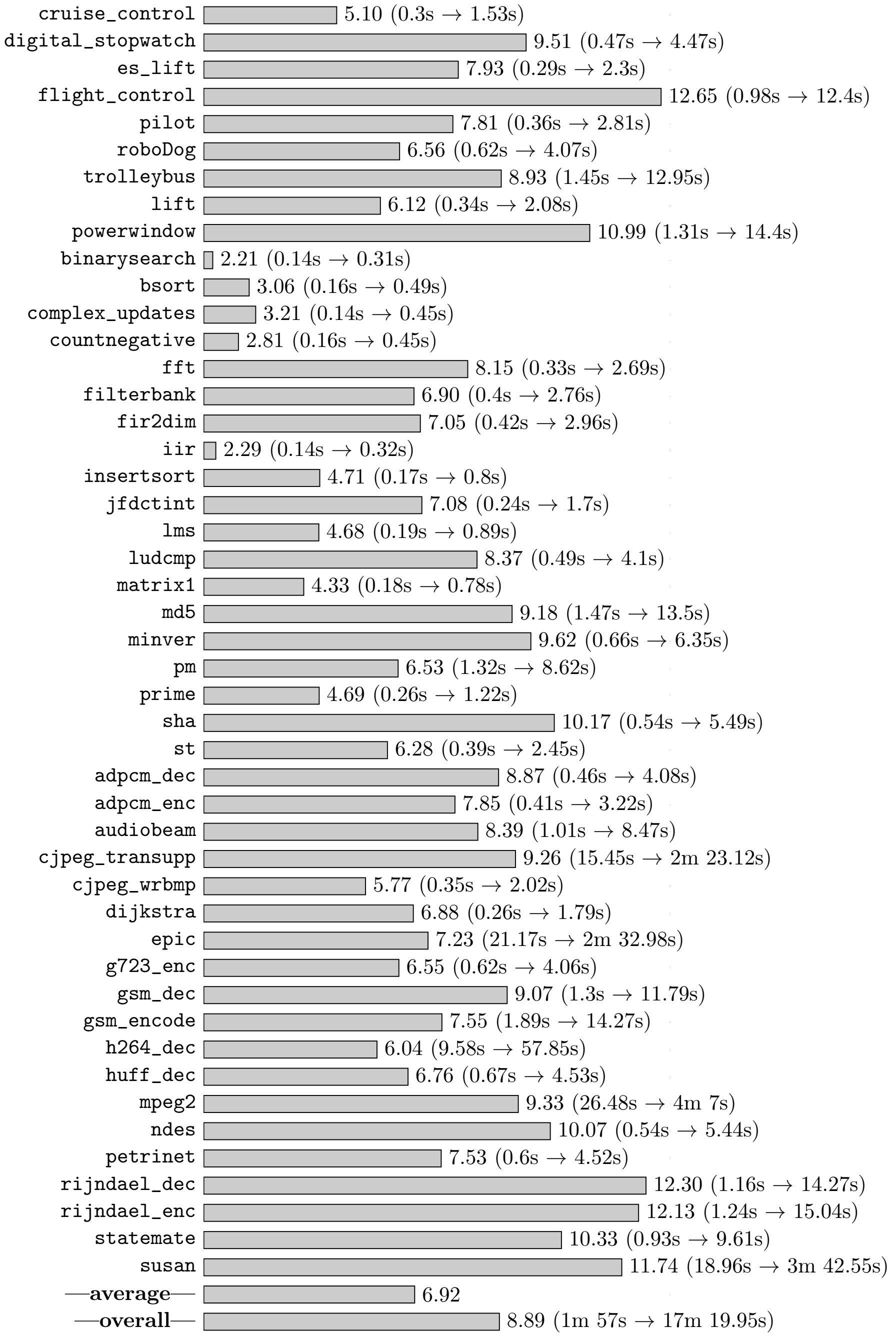


Figure 9.3.: Co-runner-insensitive WCET analysis (naive Implementation) for a quad-core processor with core configuration $Conf_{is}^{io}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

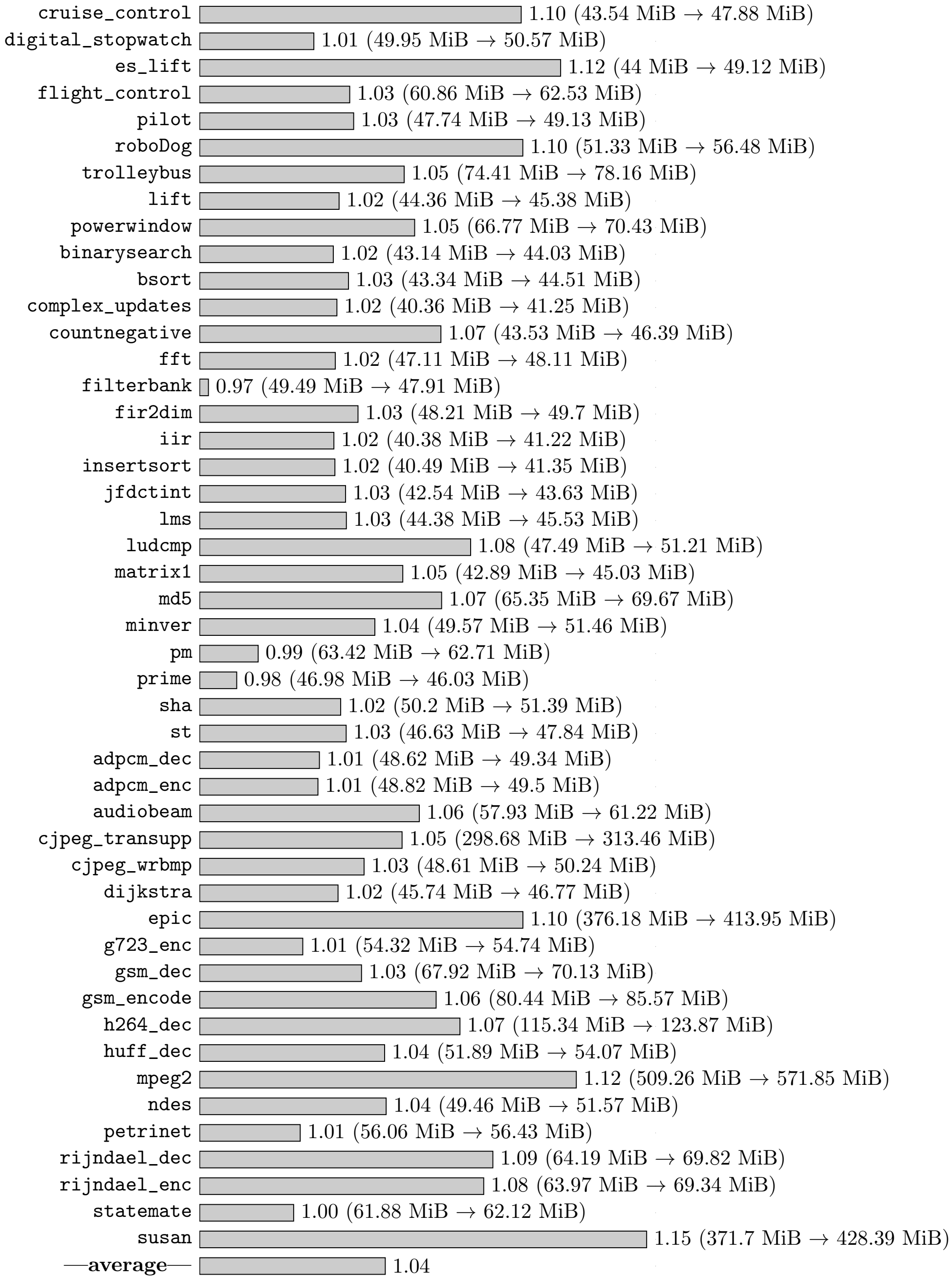


Figure 9.4.: Co-runner-insensitive WCET analysis (naive Implementation) for a quad-core processor with core configuration $Conf_{is}^{io}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

Instead, in the following two sections, we present two key implementation tricks that go beyond the naive implementation evaluated in this section. They significantly reduce the analysis runtime and, thus, make it more likely that the approach we propose scales to larger programs executed on real-world multi-core processors.

9.2. Fast-Forwarding of Converged Chains

The pipeline state of a processor core which performs an access to the shared bus often *converges* after a few pending access cycles. We call a pipeline state *converged during an access* to the shared bus if it is guaranteed to not change before the access completes. For an in-order pipeline, for example, this convergence means that the pipeline stages preceding the memory-requesting stage are filled up and the stages succeeding it have run empty. Note, however, that the pace at which a pipeline converges while accessing the shared bus inherently depends on the details of the pipeline implementation in hardware. Intuitively, pipelines with out-of-order execution are typically more successful in hiding pending bus-access cycles and, thus, typically converge later than in-order pipelines.

As our abstract model at the level of approximation of sequences of abstract states only argues about the operation of one processor core (cf. Section 7.2), the corresponding abstract state of the pipeline considered during micro-architectural analysis also often converges after a few pending access cycles. As an example for a converged abstract state, consider abstract state $\widehat{s_3}$ in Figure 9.2. The converged abstract state is guaranteed to not change before the access completes. After completion of the access (i.e. after the third granted access cycle), the abstract state of the pipeline changes to $\widehat{s_4}$.

In particular, as soon as an abstract state is converged during an access to the shared bus, further cycles of being blocked at the shared bus will not change it. As a consequence, the successors of a converged abstract state resulting from further blocked cycles do not have to be explicitly explored anymore during micro-architectural analysis. Instead, each further cycle of being blocked at the shared bus adds one cycle to the execution time in a timing-compositional manner [Hahn et al., 2013]. This implementation trick is referred to as *fast-forwarding of converged chains* [Jacobs et al., 2015]. It is sketched in Figure 9.5 for the example access of Figure 9.2. It avoids the explicit consideration of sequences of blocked cycles during which the abstract state does not change. In this way, the complexity of the micro-architectural is reduced as less case splits are performed.

Note that the fast-forwarding of converged chains is an optimization that does not change the WCET bound compared to the naive implementation (cf. Section 9.1). A formal proof of this statement, however, is beyond the scope of this thesis. We have experimentally validated this statement for our prototype implementation by showing that the optimization has not changed the WCET bounds for the benchmarks that we consider.

Our implementation of fast-forwarding, however, does not (yet) immediately detect the convergence of an abstract state based on its content. Instead, it marks an abstract state as converged if it is the result of a pending and non-completing bus access cycle transition and its predecessor state is identical to it. This means that our implementation detects the convergence of an abstract state one cycle transition after it has converged. Thus, our implementation does not (yet) exploit the full potential of fast-forwarding of converged chains.

The figures in this chapter only depict the fast-forwarding of cycles blocked at the shared bus. Our implementation, however, also uses fast-forwarding for not having to explicitly consider long chains of granted access cycles during which the abstract state does not change as it is converged. Thus, the fast-forwarding of converged chains can also reduce the complexity of single-core WCET analysis. In this context, one of our students has shown that a timing-compositional treatment of the caches during WCET analysis does not significantly reduce the analysis runtime compared to a cache modeling scheme that is integrated with the pipeline modeling and making use of

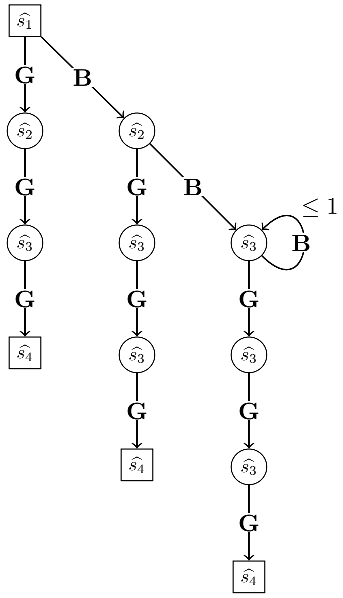


Figure 9.5.: We call an abstract state (here $\widehat{s_3}$) *converged* during an access to the shared bus if it is guaranteed to not change before the access completes. Thus, the successors of a converged abstract state resulting from further blocked cycles do not have to be explicitly explored anymore during micro-architectural analysis. Instead, each further blocked cycle adds one cycle to the execution time in a timing-compositional manner. This optimization is referred to as *fast-forwarding of converged chains*.

fast-forwarding of converged chains [Faymonville, 2015]. A detailed discussion of the benefits of fast-forwarding of converged chains during single-core WCET analysis, however, is beyond the scope of this thesis.

In order to experimentally evaluate the effectiveness of the fast-forwarding of converged chains, we use an implementation featuring this optimization for the calculation of co-runner-insensitive WCET bounds. The programs under analysis (cf. Table 9.2) are assumed to be executed on the quad-core processor sketched in Figure 9.1 with the processor core configuration $Conf_{is}^{io}$ from Table 9.1.

The diagram in Figure 9.6 shows the analysis runtime per benchmark normalized to the corresponding runtime of a single-core analysis (i.e. assuming the absence of shared-bus interference). The increase in analysis runtime is between 13 percent (for benchmark `bsort`) and 178 percent (for benchmark `ludcmp`). On average, the analysis takes 47 percent longer than an analysis assuming the absence of shared-bus interference. The corresponding average factor of 1.47 has been calculated as the geometric mean of the per-benchmark factors. The overall experiment (i.e. the analysis of all considered programs) has a runtime of three minutes and 5.16 seconds (compared to one minute and 57 seconds for a corresponding experiment assuming the absence of shared-bus interference).

Analogously, the diagram in Figure 9.7 shows the analysis memory consumption per benchmark normalized to the corresponding memory consumption of a single-core analysis (i.e. assuming the absence of shared-bus interference). The memory consumption increases by up to 15 percent (for benchmark `susan`) compared to a single-core analysis. For some of the benchmarks, e.g. for benchmark `prime`, the memory consumption even slightly decreases compared to a single-core analysis. On average, the analysis consumes three percent more memory than an analysis assuming the absence of shared-bus interference.

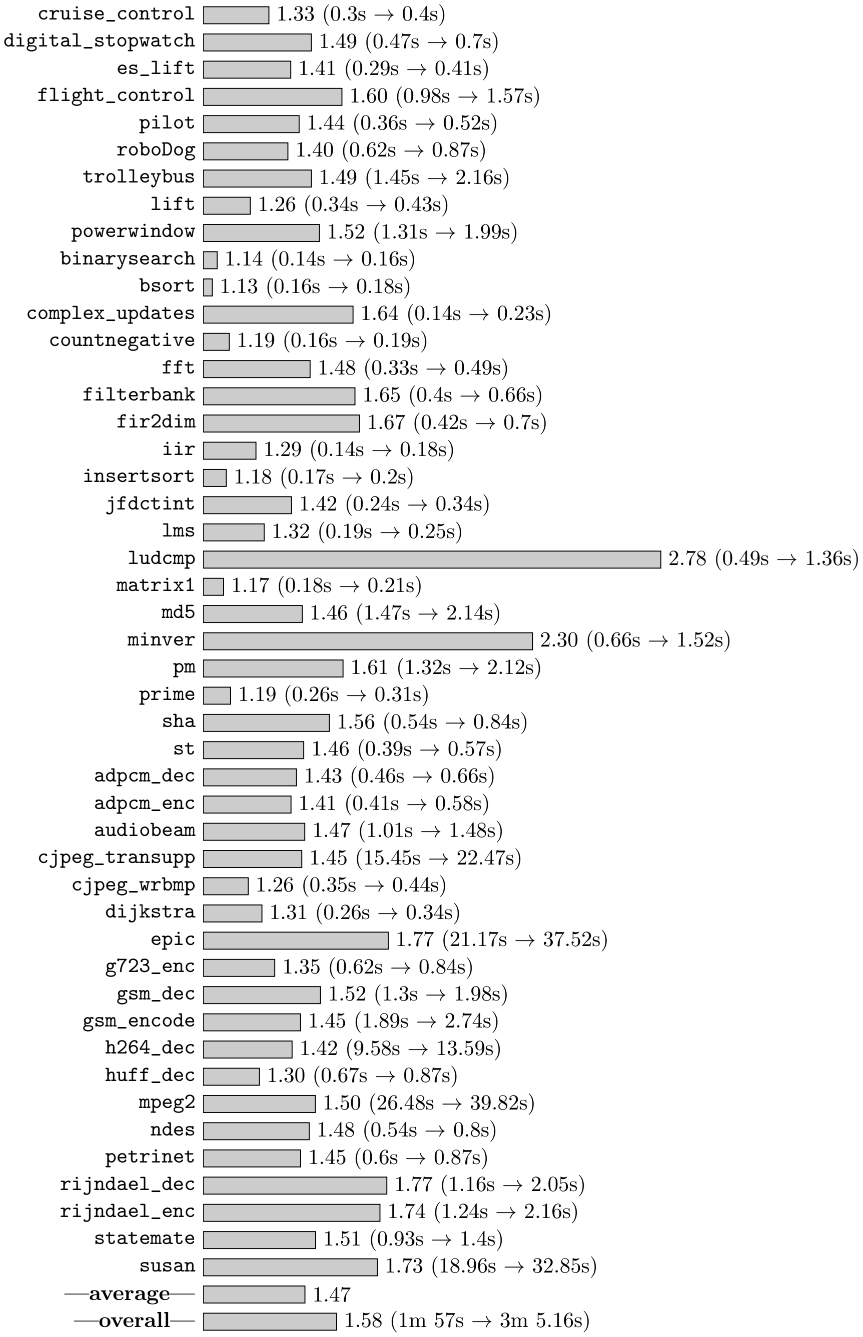


Figure 9.6.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains) for a quad-core processor with core configuration $Conf_{is}^{io}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

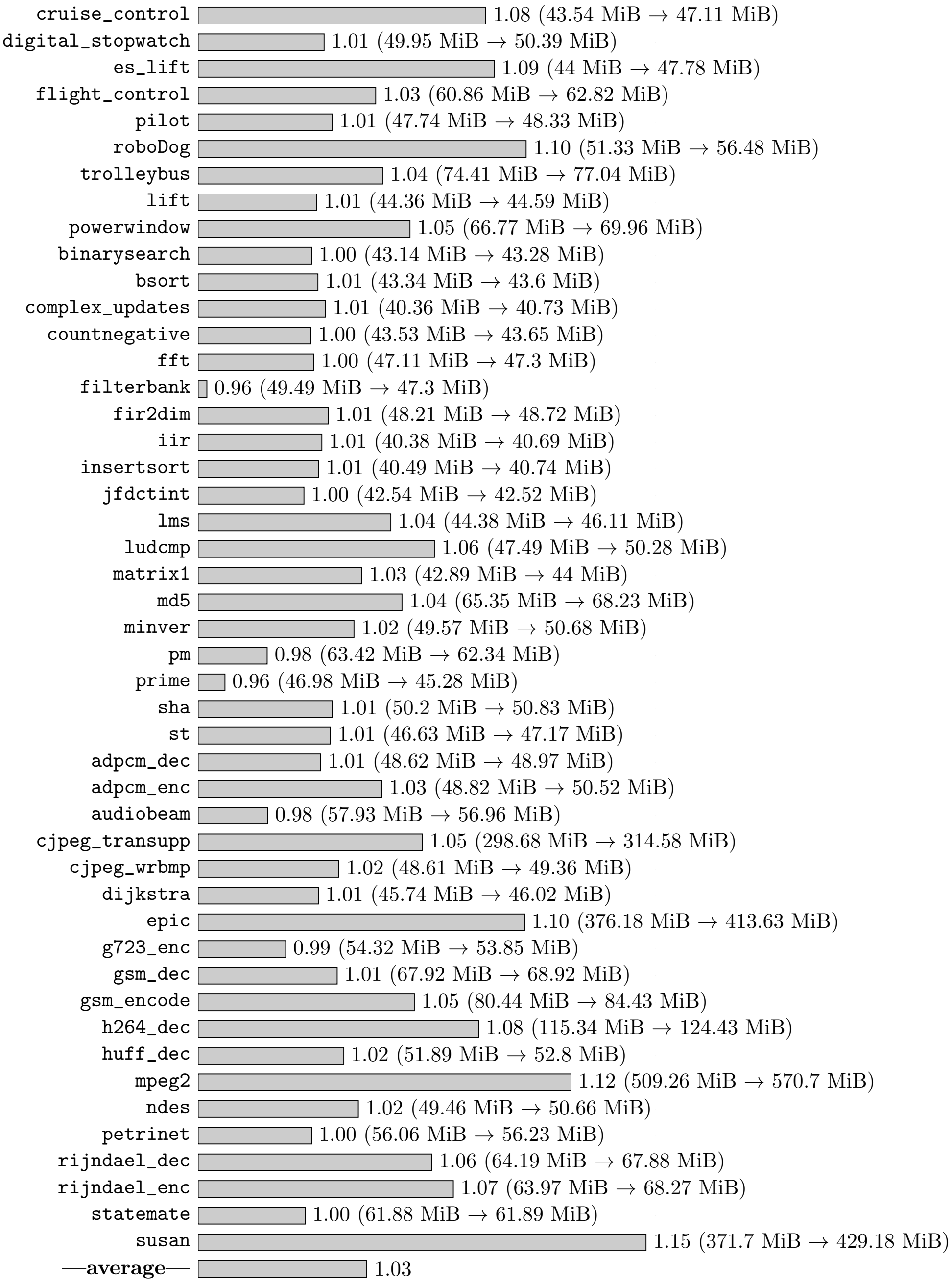


Figure 9.7.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains) for a quad-core processor with core configuration $Conf_{is}^{io}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

The results of this experiment demonstrate the effectiveness of fast-forwarding of converged chains as the average increase in analysis runtime compared to a single-core analysis is reduced to 47 percent. The naive implementation (cf. Section 9.1), in contrast, results in an average increase factor of 6.92 (i.e. an average increase of 592 percent, cf. Figure 9.3). Thus, the fast-forwarding avoids most of the runtime overhead that the naive implementation suffers from. Note that the already relatively low average increase in memory consumption achieved by the naive implementation (four percent, cf. Figure 9.4) is only slightly reduced by the fast-forwarding of converged chains (three percent).

9.3. Additionally Delaying the Case Splits

For a processor core, it does typically not matter whether an access to the shared bus is blocked for one cycle or granted for one cycle without completing. The processor core is typically only aware of a pending access cycle that does not complete the access. Thus, from a timing perspective, it does not matter whether the blocked cycles happen at the beginning or at the end of an access to the shared bus.

As a consequence, it is safe to *delay the case splits* and to perform them at the end of each access. This principle is sketched in Figure 9.8a for the example access of Figure 9.2. After the access has been granted for three cycles, it can be blocked for up to three cycles. In the sketch of Figure 9.8a, we use $\mathcal{E}$-transitions to express that the access is not blocked for one more cycle and has already completed during the preceding cycle transition.

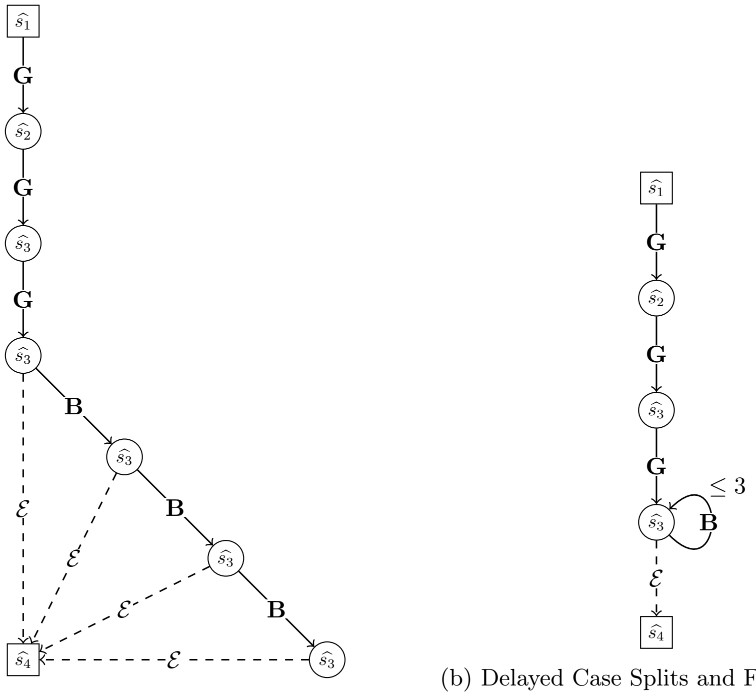
We benefit from delayed case splits if we combine them with fast-forwarding of converged chains. Delaying the case splits to the end of the access may increase the length of the converged chains of blocked cycles. This means that the micro-architectural analysis may be able to fast-forward more blocked cycles than without the delayed case splits and, thus, be more efficient. This advantage is demonstrated in Figure 9.8b for the example access of Figure 9.2: The combination of delayed case splits and fast-forwarding of converged chains avoids three case splits. Fast-forwarding without delayed case splits, in contrast, only avoids one case split (cf. Figure 9.5). Thus, delaying the case splits can be seen as a technique that supports the fast-forwarding of converged chains. Consequently, we only experimentally evaluate delayed case splits in combination with fast-forwarding.

Note that delaying the case splits is an optimization that does not change the WCET bound compared to the naive implementation (cf. Section 9.1). A formal proof of this statement, however, is beyond the scope of this thesis.

Further note that our actual implementation does not exactly delay the case splits as presented in Figure 9.8b. Instead, due to technical reasons, it delays the case splits in a way that the last granted access cycle stays at the end of the access. The combination of this variant of delaying the case splits and fast-forwarding is sketched in Figure 9.9 for the example access of Figure 9.2. As a consequence, in general, our implementation does not exploit the full potential of delaying the case splits (i.e. for some accesses, it might lead to an additional case split compared to the maximally consequent implementation presented in Figure 9.8b).

In order to experimentally evaluate the combination of delayed case splits and fast-forwarding of converged chains, we use an implementation featuring both optimizations for the calculation of co-runner-insensitive WCET bounds. The programs under analysis (cf. Table 9.2) are assumed to be executed on the quad-core processor sketched in Figure 9.1 with the processor core configuration $Conf_{is}^{io}$ from Table 9.1.

The diagram in Figure 9.10 shows the analysis runtime per benchmark normalized to the corresponding runtime of a single-core analysis (i.e. assuming the absence of shared-bus interference). The increase in analysis runtime is between zero percent (e.g. for benchmark `iir`) and 20 percent (for benchmark `minver`). On average, the analysis takes seven percent longer than an analysis assuming the absence of shared-bus interference. The corresponding average factor of 1.07 has



(a) Delayed Case Splits

(b) Delayed Case Splits and Fast-Forwarding of Converged Chains

Figure 9.8.: From a timing perspective, it does not matter whether blocked cycles happen at the beginning or at the end of a bus access. Thus, it is safe to *delay the case splits* to the end of an access (cf. Figure 9.8a). As a consequence, the micro-architectural analysis might be able to fast-forward a greater number of blocked cycles than without delayed case splits: In Figure 9.8b, this avoids three case splits compared to only one case split in Figure 9.5.

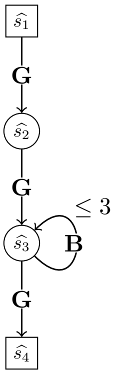


Figure 9.9.: Due to technical reasons, we implemented the combination of delayed case splits and fast-forwarding of converged chains as shown in this figure. Thus, in general, our implementation does not exploit the full potential of delaying the case splits.

been calculated as the geometric mean of the per-benchmark factors. The overall experiment (i.e. the analysis of all considered programs) has a runtime of two minutes and 7.64 seconds (compared to one minute and 57 seconds for a corresponding experiment assuming the absence of shared-bus interference).

Analogously, the diagram in Figure 9.11 shows the analysis memory consumption per benchmark normalized to the corresponding memory consumption of a single-core analysis (i.e. assuming the absence of shared-bus interference). The memory consumption increases by up to 21 percent (e.g. for benchmark `epic`) compared to a single-core analysis. For benchmark `jfdctint`, it even decreases by three percent. On average, the analysis consumes five percent more memory than an analysis assuming the absence of shared-bus interference.

The results of this experiment demonstrate the effectiveness of the combination of delayed case splits and fast-forwarding of converged chains as the average increase in analysis runtime compared to a single-core analysis is further reduced to seven percent. Thus, for the considered processor core configuration, the analysis only takes slightly longer than a single-core analysis. The fast-forwarding of converged chains alone still results in an average increase of 47 percent (cf. Figure 9.6). Note that the average increase in memory consumption of the combination of delayed case splits and fast-forwarding (five percent) is slightly higher than for fast-forwarding (three percent, cf. Figure 9.7).

We conduct the same experiment for the other processor core configurations listed in Table 9.1 (i.e. for configurations $Conf_{is}^{ooo}$, $Conf_{ic}^{io}$, and $Conf_{ic}^{ooo}$). For the detailed diagrams representing the per-benchmark increase in runtime and memory consumption of the analysis compared to a single-core analysis, we refer to pages 318 to 323 in Appendix B. Table 9.3 lists the average increase factors for runtime and memory consumption per processor core configuration. The average increase factors for processor core configurations featuring an instruction scratchpad ($Conf_{is}^{io}$ and $Conf_{is}^{ooo}$) are smaller than the average increase factors for the configurations featuring an instruction cache ($Conf_{ic}^{io}$ and $Conf_{ic}^{ooo}$). We think the reason is that, in combination with an instruction scratchpad, the majority of all memory accesses (namely the instruction accesses) is served locally by the scratchpad. Thus, only relatively few memory accesses (namely the data accesses that are not classified as cache hits) experience shared-bus interference. Hence, the relative impact of modeling the shared-bus interference on the runtime and memory consumption of the analysis is relatively small. The average increase factors for the configurations featuring an instruction cache ($Conf_{ic}^{io}$ and $Conf_{ic}^{ooo}$), in contrast, are higher because the instruction accesses of the cores with these configurations can experience shared-bus interference as well. Note, in particular, that the highest increase factors are observed for the processor core configuration featuring out-of-order execution and an instruction cache. Intuitively, complex processor core features (in our case out-of-order execution and instruction caches) interact in order to hide as much memory latency as possible. Consequently, the state of the processor core pipeline typically converges later for more complex processor cores. As the presented optimizations exploit the convergence of the processor core pipeline, they are less effective for more complex processor cores and, thus, the overhead in terms of analysis runtime and memory consumption increases with the complexity of the processor cores (cf. Table 9.3). This is in line with recent results [Hahn et al., 2016b] showing an even higher overhead for processor cores additionally featuring a store buffer. The consideration of processor core configurations featuring a store buffer, however, is beyond the scope of this thesis.

We also conduct the same series of experiments for the octa-core processor sketched in Figure 9.12. For the detailed diagrams representing the per-benchmark increase in runtime and memory consumption of the analysis compared to a single-core analysis, we refer to pages 324 to 331 in Appendix B. Table 9.4 lists the average increase factors for runtime and memory consumption per processor core configuration. Note that these average increase factors are almost identical to those of the quad-core processor (cf. Table 9.3). Thus, the co-runner-insensitive WCET analysis for the considered octa-core processors is essentially not more complex than the corresponding analysis for the considered quad-core processors. Intuitively, for all considered

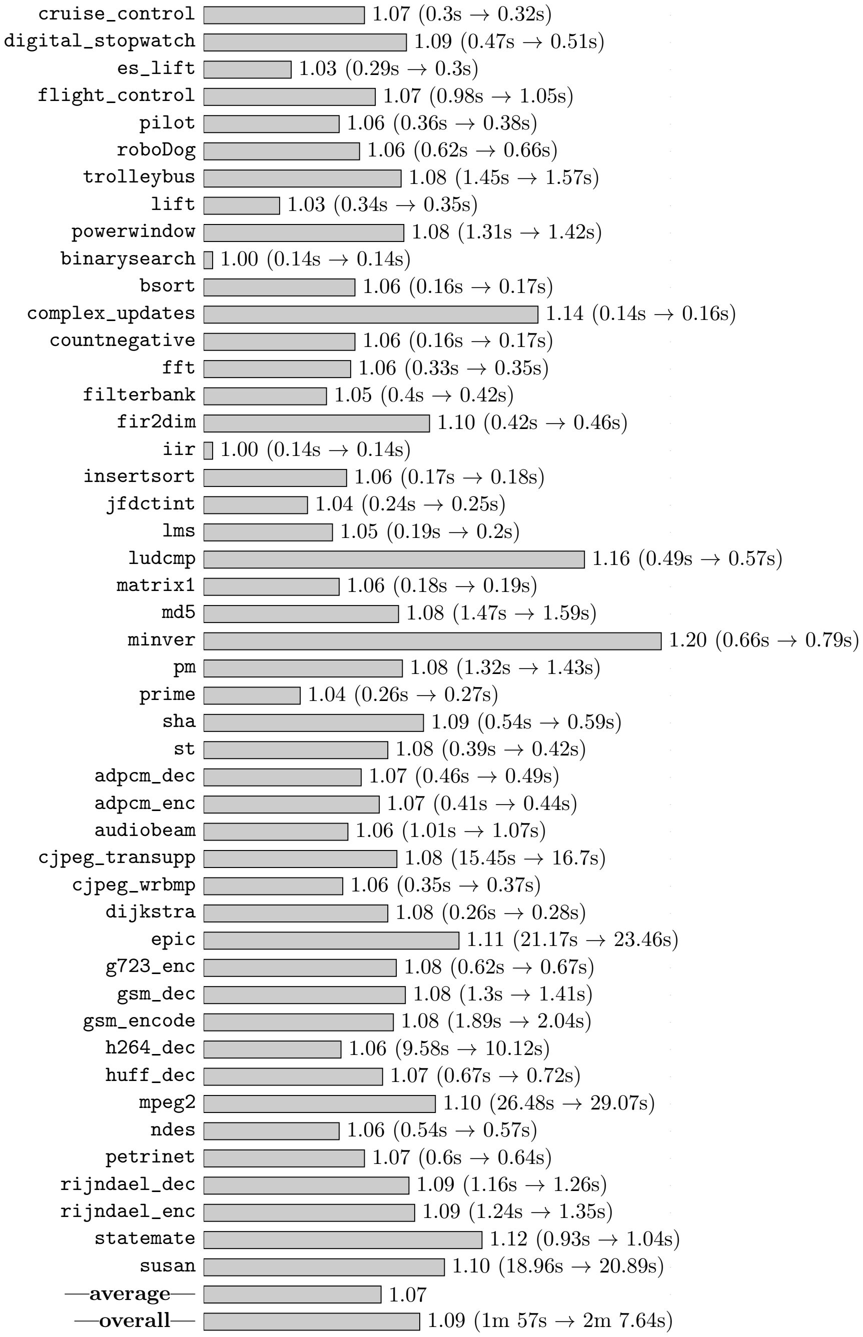


Figure 9.10.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{is}^{io}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

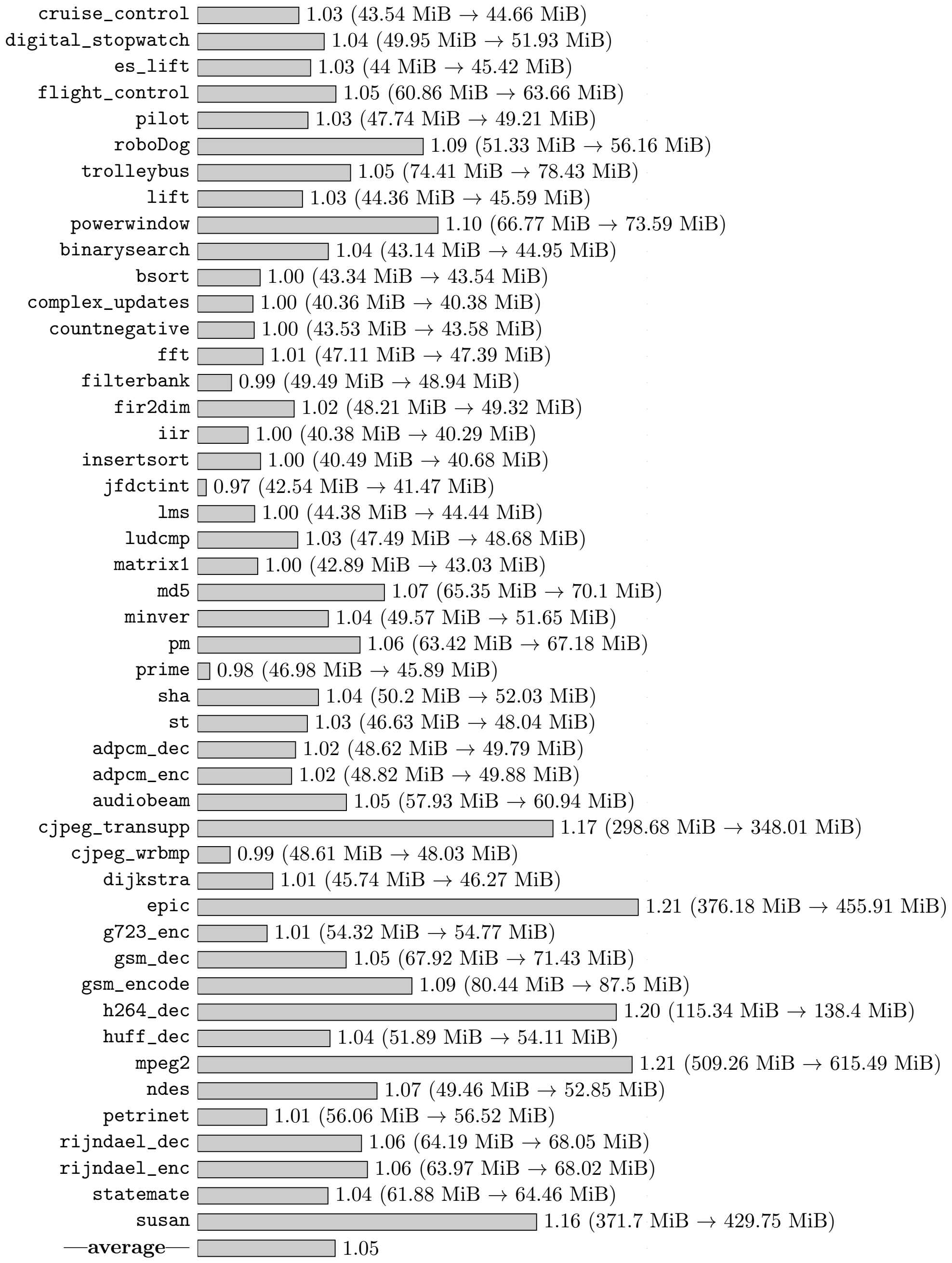


Figure 9.11.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{is}^{io}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

	Average Increase in	
	Runtime	Memory Consumption
$Conf_{is}^{io}$	1.07	1.05
$Conf_{is}^{ooo}$	1.10	1.06
$Conf_{ic}^{io}$	1.13	1.10
$Conf_{ic}^{ooo}$	1.15	1.12

Table 9.3.: Average increase factors for runtime and memory consumption per processor core configuration for the co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a *quad-core* processor.

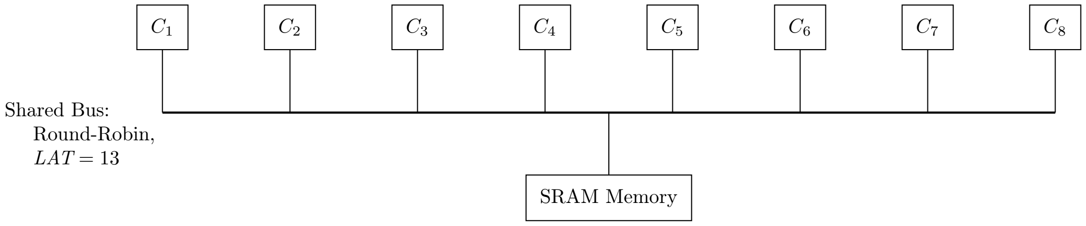


Figure 9.12.: We additionally evaluate the impact of the number of processor cores. To this end, we consider an octa-core processor with a shared bus, round-robin bus arbitration, and an SRAM memory. We assume a fixed latency of 13 clock cycles per granted bus access.

processor core configurations and all accesses to the shared bus, the state of the pipeline already converges for a quad-core processor. As a consequence, thanks to the presented optimizations, all additional blocked cycles that occur on an octa-core processor are fast-forwarded and do not further increase the runtime or the memory consumption of the analysis. This means that the presented implementation of our analysis is *scalable* in the sense that—once the state of the pipeline converges for every access to the shared bus—increasing the number of processor cores does not further increase the analysis complexity.

Note that the processor cores used in real-world multi-core processors are typically significantly more complex than the processor core configurations that we consider (cf. Table 9.1). The implementation of our analysis approach for a real-world multi-core processor, however, is beyond the scope of this thesis. We expect that our industry partner[2] will investigate to which real-world

[2] https://www.absint.com

	Average Increase in	
	Runtime	Memory Consumption
$Conf_{is}^{io}$	1.07	1.05
$Conf_{is}^{ooo}$	1.10	1.06
$Conf_{ic}^{io}$	1.12	1.11
$Conf_{ic}^{ooo}$	1.14	1.12

Table 9.4.: Average increase factors for runtime and memory consumption per processor core configuration for the co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an *octa-core* processor.

multi-core platforms our analysis approach is applicable in terms of analysis complexity. In particular, such an investigation will have to answer the following questions per multi-core platform.

- For a given number of processor cores, is the average increase in analysis runtime and memory consumption compared to a single-core analysis still considered manageable?
- What is the smallest number of processor cores starting from which we can add more cores to the system without further increasing the analysis runtime and memory consumption for most programs?

Relating the Presented Implementation Tricks to an Assumption that Timing-Compositional Analyses Typically Rely on For a hardware platform on which the pipeline state of the processor core is guaranteed to converge within the granted access latency for every bus access, additional cycles of being blocked at the shared bus are guaranteed to not trigger any timing anomalies. In a recent paper [Wegener, 2017], the author argues that the pipeline state of the processor core typically converges within the granted access latency for every bus access. In the remainder of this thesis, we refer to hardware platforms for which this is guaranteed as fulfilling *Wegener's assumption*. For any hardware platform which fulfills Wegener's assumption, it is safe to perform a single-core WCET analysis (i.e. assuming the absence of shared-bus interference) and to subsequently add an upper bound on the possible numbers of blocked cycles in a timing-compositional way. However, in general, it remains completely unclear how to prove Wegener's assumption for a given hardware platform [Wegener, 2017].

Our analysis approach (i.e. modeling shared-bus interference by non-determinism, cf. Chapter 7), in contrast, is sound for arbitrary hardware platforms and, thus, its soundness does not inherently rely on Wegener's assumption. Nevertheless, the optimized implementation of our approach also profits from Wegener's assumption. If the pipeline state of the processor core is guaranteed to converge within the granted access latency for every bus access, an implementation that exploits the full potential of delaying the case splits and fast-forwarding of converged chains will always fast-forward all blocked cycles and, thus, essentially not be more complex than an analysis ignoring the shared-bus interference. Note, however, that—as described above—our implementation does not exploit the full potential of the presented implementation tricks. Nonetheless, we are confident that our implementation is not significantly more complex than an analysis ignoring the shared-bus interference in case the pipeline state of the processor core converges within the granted access latency for most of the bus accesses (i.e. Wegener's assumption holds for most of the bus accesses). As a consequence, if our implementation is significantly more complex than an analysis ignoring the shared-bus interference, this strongly indicates that Wegener's assumption does not hold for the considered hardware platform and, thus, the combination of a single-core analysis and a timing-compositional post-processing is not suitable for the timing verification of safety- or mission-critical applications executed on the considered hardware platform.

We only recommend the combination of a single-core analysis and a timing-compositional post-processing for the timing verification of a safety- or mission-critical application if the hardware platform on which the application is executed does not feature timing anomalies which are triggered by shared-bus interference (i.e. by cycles blocked at the shared bus). To the best of our knowledge, there are currently only two approaches that guarantee the absence of this particular type of timing anomaly. The first approach relies on proving that Wegener's assumption holds for the considered hardware platform. As mentioned before, it is still unclear how to conduct such a proof for a given hardware platform. The second approach relies on a custom hardware modification which stalls the pipeline of the processor core under analysis on every cycle of being blocked at the shared bus [Hahn et al., 2016a]. In case the absence of this particular type of timing anomaly is not guaranteed (i.e. none of the aforementioned two approaches is applicable), we recommend modeling the shared-bus interference by non-determinism (cf. Chapter 7) and making use of the implementation tricks presented in this chapter (i.e. fast-forwarding and delayed case splits). Either this approach is not significantly more complex than a single-core analysis, or a significant increase in complexity is a strong indication that Wegener's assumption does anyway not hold for the considered hardware platform. Nonetheless, for the timing verification of less critical applications, it may provide a sufficient degree of confidence to rely on Wegener's assumption without proving that it holds.

9.4. Implementing Implicit Path Enumeration without Binary Variables

Classically, implicit path enumeration has been described as a pure flow problem (i.e. one integer variable per edge and in-flow equals out-flow per node) [Li and Malik, 1995; Puschner and Schedl, 1997]. The variant of implicit path enumeration that we formalize in Section 5.4, in contrast, features additional binary variables describing which edge starts respectively ends the implicit path.

The implicit path enumerations calculating the WCET bounds for all experiments in this chapter (naive implementation, fast-forwarding, and fast-forwarding with delayed case splits) have been implemented without binary variables. In this section, we argue that the implementation without binary variables is a safe overapproximation (i.e. an abstract model) of the implicit path enumeration formalized in Section 5.4.

In the following, let G^X be the graph on which the implicit path enumeration with binary variables (cf. Section 5.4) operates.

$$G^X = (Nodes^X, Nodes^X_{start}, Nodes^X_{end}, Edges^X) \tag{9.1}$$

Implicit path enumeration without binary variables, in contrast, shall operate on graph G^Y, which is a slightly modified version of graph G^X.

$$G^Y = (Nodes^Y, Nodes^Y_{start}, Nodes^Y_{end}, Edges^Y) \tag{9.2}$$

In addition to the nodes of G^X, graph G^Y shall contain a fresh dummy node dm.

$$Nodes^Y = Nodes^X \cup \{dm\} \tag{9.3}$$

The dummy node dm shall be the only start node and the only end node of graph G^Y.

$$Nodes^Y_{start} = \{dm\} \tag{9.4}$$

$$Nodes^Y_{end} = \{dm\} \tag{9.5}$$

In addition to the edges of G^X, graph G^Y shall have an outgoing edge from dm to every start node of G^X and an incoming edge to dm from every end node of G^X.

$$Edges^Y = Edges^X \cup \{(dm, nd) \mid nd \in Nodes^X_{start}\} \cup \{(nd, dm) \mid nd \in Nodes^X_{end}\} \tag{9.6}$$

The weights of the edges of graph G^X shall be reused in graph G^Y. All additional edges of graph G^Y shall have zero-valued weights.

$$\forall e \in Edges^Y \setminus Edges^X : \forall E \in Events : \widehat{wE^{UB}}(e) = 0 \land \widehat{wE^{LB}}(e) = 0 \tag{9.7}$$

Based on graph G^Y, we specify an implicit path enumeration that does not rely on binary variables. It only keeps track of how often each edge is contained in a considered implicit path.

$$\begin{aligned}\widehat{Implicit'} = \{ & timesTaken : Edges^Y \to \mathbb{N} \mid \\ & \sum_{e \in Edges^Y_{start}} timesTaken(e) = 1 \land \\ & [\ \forall node \in Nodes^Y : \\ & \quad \sum_{e_{in} \in inEdges(node)} timesTaken(e_{in}) = \sum_{e_{out} \in outEdges(node)} timesTaken(e_{out})\ ] \\ \} & \end{aligned} \tag{9.8}$$

The function γ'_{impli} maps an implicit path without binary variables to a set of implicit paths with binary variables.

$$\gamma'_{impli} : \widehat{Implicit'} \to \mathcal{P}(\widehat{Implicit}) \tag{9.9}$$

$$\begin{aligned}\gamma'_{impli}(timesTaken^Y) = \{ & (timesTaken^X, *, *) \in \widehat{Implicit} \mid \\ & \forall e \in Edges^X : timesTaken^X(e) = timesTaken^Y(e)\}\end{aligned} \tag{9.10}$$

Intuitively, $(\widehat{Implicit'}, \gamma'_{impli})$ is an overapproximation of $\widehat{Implicit}$. A formal proof of this statement is omitted due to time and space constraints.

$$\bigcup_{\widehat{i'} \in \widehat{Implicit'}} \gamma'_{impli}(\widehat{i'}) \supseteq \widehat{Implicit} \tag{9.11}$$

Consequently, $(\widehat{Implicit'}, \gamma'_{impli})$ is also an overapproximation of the subset of $\widehat{Implicit}$ for which all the lifted system properties hold.

$$\bigcup_{\widehat{i'} \in \widehat{Implicit'}} \gamma'_{impli}(\widehat{i'}) \supseteq \{\widehat{i} \in \widehat{Implicit} \mid \forall P_k \in Prop : \widehat{P_k^{impli}}(\widehat{i})\} \tag{9.12}$$

Thus, we can once more apply property lifting to further lift the properties $\widehat{P_k^{impli}}$ to the level of implicit path enumeration without binary variables. The properties lifted to implicit path enumeration without binary variables are annotated with the superscript *impli'*. Consequently, we rephrase soundness criterion (4.C1) in the following way for this step of further lifting an already lifted property.

$$\forall \widehat{i'} \in \widehat{Implicit'} : [\, \exists \widehat{i} \in \gamma'_{impli}(\widehat{i'}) : \widehat{P_k^{impli}}(\widehat{i})\,] \Rightarrow \widehat{P_k^{impli'}}(\widehat{i'}) \tag{9.C1}$$

Intuitively, if the definition of a property $\widehat{P_k^{impli}}$ does syntactically not contain any of the functions *isStart* and *isEnd*, the property can safely be reused at the level of implicit path enumeration without binary variables as it is guaranteed to fulfill the lifting criterion (9.C1). Note that a formal proof of this statement is beyond the scope of this thesis. We refer to such a safe reuse of a property as *trivial lifting*.

$$\begin{aligned}& \widehat{P_k^{impli}} \text{ does syntactically not contain any of the functions } isStart \text{ and } isEnd \\ \Rightarrow\ & \forall \widehat{i'} \in \widehat{Implicit'} : [\, \exists \widehat{i} \in \gamma'_{impli}(\widehat{i'}) : \widehat{P_k^{impli}}(\widehat{i})\,] \Leftrightarrow \widehat{P_k^{impli}}(\widehat{i'})\end{aligned} \tag{9.13}$$

The majority of control flow properties typically taken into account during implicit path enumeration (as e.g. loop-bounding properties) only argue about which edge of the graph is taken how often. Thus, such properties can be trivially lifted to an implicit path enumeration without binary variables. More involved system properties (as e.g. the property upper-bounding the number of blocked cycles in a co-runner-sensitive analysis starting from an optimistic initialization, cf. equations (7.37) and (7.38)), however, also argue about which edge starts respectively ends the currently considered graph path. For such properties, it may be necessary to explicitly lift them from implicit path enumeration with binary variables to implicit path enumeration without binary variables in a way that criterion (9.C1) is fulfilled.

In addition to lifting the system properties, for a safe bound calculation, we have to make sure that the objective of the implicit path enumeration without binary variables safely approximates the objective of the implicit path enumeration with binary variables. To this end, we rely on additional criteria for safely *lifting the objective* from implicit path enumeration with binary variables to implicit path enumeration without binary variables. In case the *objective is maximized*, the objective value per implicit path without binary variables has to be at least as high as the objective value of every implicit path with binary variables that it describes.

$$\forall \widehat{i'} \in \widehat{Implicit'} : \forall \widehat{i} \in \gamma'_{impli}(\widehat{i'}) : \widehat{Obj'}(\widehat{i'}) \geq \widehat{Obj}(\widehat{i}) \tag{9.C2}$$

In case the *objective is minimized*, the objective value per implicit path without binary variables must not exceed the objective value of any implicit path with binary variables that it describes.

$$\forall \widehat{i'} \in \widehat{Implicit'} : \forall \widehat{i} \in \gamma'_{impli}(\widehat{i'}) : \widehat{Obj'}(\widehat{i'}) \leq \widehat{Obj}(\widehat{i}) \tag{9.C3}$$

Intuitively, if the definition of the objective $\widehat{Obj}$ does syntactically not contain any of the functions *isStart* and *isEnd*, the objective $\widehat{Obj}$ can safely be reused at the level of implicit path enumeration without binary variables as it is guaranteed to fulfill the lifting criterion (9.C2) respectively (9.C3). Note that a formal proof of this statement is beyond the scope of this thesis. As above, we refer to such a safe reuse of an objective as *trivial lifting.*

$$\begin{aligned}&\widehat{Obj} \text{ does syntactically not contain any of the functions } \mathit{isStart} \text{ and } \mathit{isEnd}\\ \Rightarrow\ &\forall \widehat{i'} \in \widehat{Implicit'} : \forall \widehat{i} \in \gamma'_{impli}(\widehat{i'}) : \widehat{Obj}(\widehat{i'}) = \widehat{Obj}(\widehat{i})\end{aligned} \tag{9.14}$$

Classically, the objective used in implicit path enumeration only argues about which edge of the graph is taken how often (cf. equations (6.173) and (6.174)) [Li and Malik, 1995; Puschner and Schedl, 1997]. More involved applications of implicit path enumeration (as e.g. the calculation of compositional base bounds for schedulability analyses starting from an optimistic initialization, cf. equations (8.16) and (8.17)), however, also argue about which edge starts respectively ends the considered implicit path. For such applications, it may be necessary to explicitly lift the objective from implicit path enumeration with binary variables to implicit path enumeration without binary variables in a way that criterion (9.C2) respectively criterion (9.C3) is fulfilled.

In case every lifted property and the objective used at the level of implicit path enumeration with binary variables can be trivially lifted to the level of implicit path enumeration without binary variables, the whole implicit path enumeration can trivially be lifted to an implementation without binary variables. Such a trivial lifting of the whole implicit path enumeration is guaranteed to result in the same bound as the original implicit path enumeration with binary variables. Note that a formal proof of this statement is beyond the scope of this thesis. In general, however, an implementation without binary variables can result in less precise bounds than an implementation with binary variables. An example demonstrating such loss of precision is omitted due to time and space constraints.

The implicit path enumerations for all experiments in this chapter can be trivially lifted to an implementation without binary variables. Thus, all experiments presented in this chapter rely on an implicit path enumeration without binary variables. A comparison of analysis runtime and

	Quad-Core	Octa-Core
$Conf_{is}^{io}$	3.13	5.95
$Conf_{is}^{ooo}$	3.19	6.10
$Conf_{ic}^{io}$	3.36	6.50
$Conf_{ic}^{ooo}$	3.47	6.75

Table 9.5.: Average increase factors of the co-runner-insensitive WCET bounds compared to WCET bounds calculated by a single-core analysis.

memory consumption between implementations with and without binary variables is beyond the scope of this chapter. Note, however, that the following chapter compares both flavors of implicit path enumeration with respect to their runtime and memory consumption during the calculation of values on arrival curves (cf. Section 10.2.3 and Section 10.2.4).

9.5. Co-Runner-Insensitive WCET Bounds

So far, we only discussed the runtime and memory consumption of different implementation variants of the proposed co-runner-insensitive WCET analysis. In this section, we take a closer look at the co-runner-insensitive WCET bounds. Note that the different implementation variants presented in this chapter (naive, fast-forwarding, and fast-forwarding with delayed splits) do not differ with respect to the WCET bounds they calculate. Thus, the results presented in this section hold for all three implementation variants.

We consider the co-runner-insensitive WCET bounds for the case in which the programs under analysis (cf. Table 9.2) are executed on the quad-core processor sketched in Figure 9.1 with the processor core configuration $Conf_{is}^{io}$ from Table 9.1. The diagram in Figure 9.13 shows the WCET bounds per benchmark normalized to the corresponding WCET bounds calculated by a single-core analysis (i.e. assuming the absence of shared-bus interference). The co-runner-insensitive WCET bounds are between 1.65 times (for benchmark `prime`) and 3.68 times (for benchmark `insertsort`) as high as the corresponding single-core WCET bounds. On average (geometric mean), they are 3.13 times as high as the single-core WCET bounds. The variance of the per-benchmark factors is due to the different densities of accesses to the shared bus on the worst-case paths of the benchmarks.

For the corresponding diagrams of the other processor core configurations and of all processor core configurations in combination with an octa-core processor, we refer to pages 332 to 338 in Appendix B. Table 9.5 lists the average increase factors of the co-runner-insensitive WCET bounds compared to single-core WCET bounds for all considered processor core configurations in combination with a quad-core and an octa-core processor.

The average increase factors for processors with out-of-order execution are always slightly higher than for the corresponding processors with in-order execution. The reason for this is that, in the single-core analysis, out-of-order execution can often hide significantly more granted bus-access cycles than in-order execution while, in the multi-core case, both out-of-order and in-order execution can typically not hide many additional blocked cycles because the pipeline already converges within the granted access cycles for most accesses (i.e. Wegener's assumption holds for most accesses). Thus, for some benchmarks, the single-core bound for out-of-order execution is significantly smaller than the single-core bound for in-order execution while the multi-core bounds for both pipeline types roughly suffer from the same amount of additional cycles due to shared-bus interference. As a consequence, for these benchmarks, the increase factor from single-core bound to multi-core bound is significantly higher in combination with out-of-order execution.

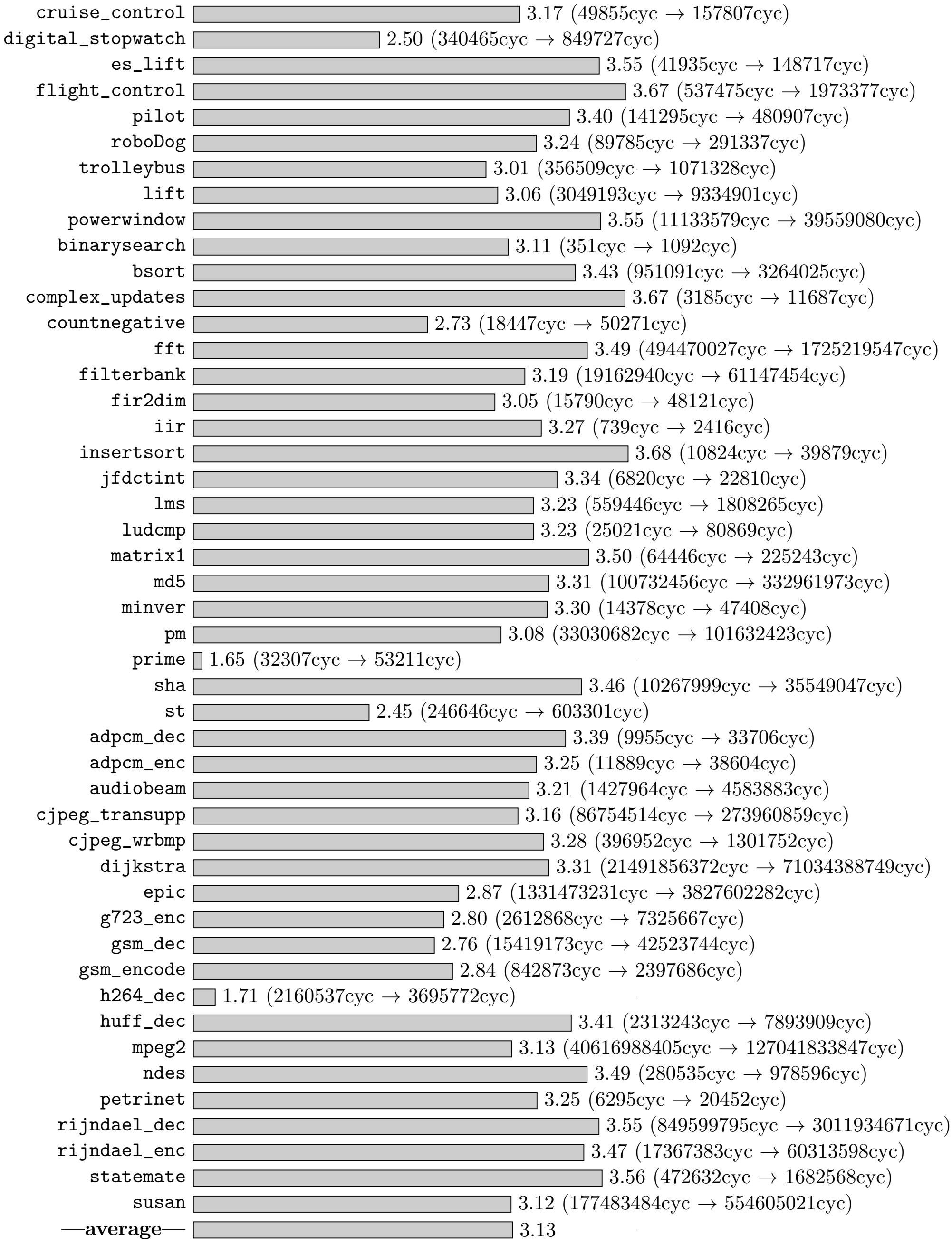


Figure 9.13.: Co-runner-insensitive WCET analysis for a quad-core processor with core configuration $Conf_{is}^{io}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

The average increase factors for processors with instruction caches are also always slightly higher than for the corresponding processors with instruction scratchpads. This is due to the greater number of bus accesses in the instruction cache setting compared to an instruction scratchpad: every miss in the instruction cache results in an additional bus access that does not exist for the scratchpad setting. Thus, the single-core bound for the instruction-cache setting is slightly greater than the single-core bound for the scratchpad setting (additional granted cycles of the instruction-cache misses) while the multi-core bound for the instruction-cache setting is significantly greater than the multi-core bound for the scratchpad setting (additional blocked cycles of the instruction-cache misses). As a consequence, the relative increase factors from a single-core bound to a multi-core bound are higher for the instruction-cache setting.

For all settings considered in Table 9.5, the average increase factors of the co-runner-insensitive WCET bounds compared to single-core WCET bounds are relatively close to the number of processor cores in the system. This is due to the co-runner-insensitive analysis implicitly assuming maximally interfering programs on the concurrent processor cores (i.e. every bus access takes up to number-of-cores times as long as without interference, cf. equation (7.13)). As the execution time of the considered programs is often dominated by memory accesses, the increase of the co-runner-insensitive WCET bounds compared to the single-core WCET bounds is relatively close to the number of processor cores. In earlier experiments with compiler optimizations disabled for the programs under analysis, the average increase factors were even closer to the number of processor cores as the execution time of most non-optimized programs is completely dominated by memory accesses. Note that all experiments presented in this thesis are conducted for programs that are compiled with compiler optimizations enabled.

On most concrete systems (i.e. in most actual settings of co-running programs), however, it cannot happen that every access to the shared bus is blocked for the theoretical maximum of cycles that is possible on the hardware platform (cf. equation (7.13)). As a consequence, the co-runner-insensitive WCET bounds are typically very imprecise. With respect to precision, a co-runner-sensitive consideration—taking into account the bus access behavior of the programs executed in parallel—is more desirable. An exact consideration of the shared-bus interference, however, would require the enumeration of all interleavings of accesses to the shared bus by the different processor cores and, thus, suffer from an unmanageable computational complexity (cf. Section 2.2.1). In order to avoid this complexity, most existing approaches to the timing verification of multi-core processors are processor-core-modular in the sense that they only consider one processor core at a time (cf. Section 2.2.2). This processor-core-modular consideration of the shared-bus interference happens either during a co-runner-sensitive WCET analysis [Jacobs et al., 2015] (cf. Section 7.5) or during schedulability analysis [Altmeyer et al., 2015]. Most existing processor-core-modular approaches rely on the *calculation of values on arrival curves*. So does also our co-runner-sensitive WCET analysis presented in Section 7.5. In the context of shared-bus interference, a value on an arrival curve is an upper bound on the numbers of accesses (respectively access cycles) that a given processor core can be granted within any time interval of at most a given length. In the next chapter, we discuss the calculation of values on arrival curves.

Chapter 10 Calculation of Values on Arrival Curves

Yet, the presented approach lacks formalisms for critical aspects to ensure safeness (i.e., the resulting arrival curve should not be underapproximated) and tightness (i.e., the level of overapproximation due to the model should be minimal).

(A recent paper published by Oehlert et al. at ECRTS 2018 referring to a contribution of our existing work [Jacobs et al., 2015], which it essentially copies without making major additional contributions. Ironically, their paper does not feature formal proofs with respect to soundness or tightness. In this context, we consider such a statement as poor scientific practice and, thus, do not further discuss the paper in this thesis.)

Co-runner-insensitive WCET analyses assume that the program under analysis experiences the maximum amount of shared-resource interference that is possible on the considered hardware platform. This means that co-runner-insensitive analyses implicitly assume maximally interfering programs to be executed on the concurrent processor cores. For most real-world systems with multi-core processors, however, the actual co-running programs are not able to generate this maximum amount of interference. As a consequence, co-runner-insensitive WCET bounds tend to be very imprecise for most real-world scenarios.

This inherent imprecision can be avoided by a *co-runner-sensitive* consideration—taking into account the access behavior of the co-running programs with respect to the shared resources. An enumeration of all interleavings of accesses to the shared bus by the different processor cores, however, suffers from an unmanageable computational complexity (cf. Section 2.2.1). In order to avoid this complexity, most co-runner-sensitive approaches are *processor-core-modular* in the sense that they only consider one processor core at a time (cf. Section 2.2.2). There are two categories of processor-core-modular approaches:

- Co-runner-sensitive WCET analysis

- Interference-aware schedulability analysis

The *co-runner-sensitive WCET analyses* [Jacobs et al., 2015] (cf. Section 7.5) perform the processor-core-modular consideration of the shared-resource interference already during the calculation of the WCET bound. The main advantage of this approach is that it can be combined with standard schedulability analyses that are not aware of shared-resource interference. However, the separate consideration of the shared-resource interference per interfered program may lead to the consideration of an infeasible overall amount of interference across multiple program runs (cf. Section 7.9).

Interference-aware schedulability analyses [Schliecker and Ernst, 2010; Altmeyer et al., 2015], in contrast, consider the overall amount of shared-resource interference that a scheduled sequence of programs can experience. All of these interference-aware schedulability analyses rely on the

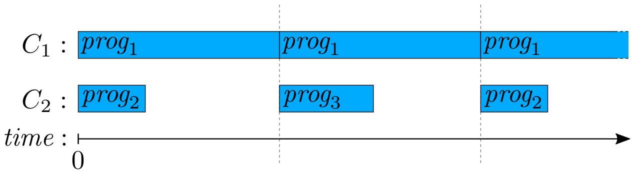


Figure 10.1.: Scenario in which the scheduler guarantees that every execution run of the considered program can overlap with at most one program execution run per concurrent processor core. It enables a simple calculation of an upper bound on the amount of interference that a program suffers from. However, such restricted scheduling scenarios can result in a poor utilization of the processing resources.

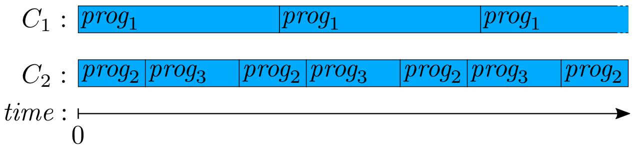


Figure 10.2.: Less restricted scheduling scenario: the amount of interference that a program execution run experiences depends on its execution time.

principle of timing compositionality [Hahn et al., 2013]. The use of compositional base bounds (cf. Chapter 8) makes these schedulability analyses applicable to hardware platforms exhibiting timing anomalies [Lundqvist and Stenstrom, 1999].

To the best of our knowledge, all existing processor-core-modular approaches to timing verification rely on upper bounds on the amount of interference that a particular processor core can generate. The amount of shared-bus interference is quantified as number of granted bus-access cycles and/or number of granted bus accesses (cf. Section 7.6). The amount of shared-cache interference is quantified as number of accesses to the shared cache [Nagar, 2016].

In case the scheduler guarantees that every execution run of the considered program can overlap with at most one program execution run per concurrent processor core (as e.g. implicitly assumed in [Nagar, 2016]), the interference bound per concurrent processor core can be calculated by the same method that is used for the calculation of a WCET bound (cf. Chapter 6). Figure 10.1 sketches such a restricted scheduling scenarios. Program $prog_1$ is executed on processor core C_1 while programs $prog_2$ and $prog_3$ are executed on processor core C_2. The scheduler guarantees that each program execution run on one of the cores can only overlap with one program execution run on the other core. Thus, an execution run of program $prog_1$ can at most experience the amount of interference that is generated by one execution run of program $prog_2$ or program $prog_3$. An upper bound on the amount of interference that a program generates during one execution run can be calculated with the techniques presented in Chapter 6. Figure 10.1, however, also demonstrates that such restricted scheduling scenarios can result in a poor utilization of the processing resources.

In a less restricted scheduling scenario, the amount of interference that a program execution run experiences depends on its execution time. This is demonstrated by the example in Figure 10.2. Since program $prog_1$ is relatively long-running, one of its execution runs can overlap with up to four program execution runs on core C_2. An execution run of the relatively short-running programs $prog_2$ or $prog_3$, in contrast, only overlaps with a relatively small portion of up to two execution runs of program $prog_1$. Thus, the amount of interference that an execution run of program $prog_2$ or $prog_3$ experiences is likely significantly smaller than the amount of interference that a complete execution run of program $prog_1$ generates.

In fact, there is a *circular dependency* between the amount of interference that a program execution run experiences and its execution time. As explained before, the execution time determines how much interference the concurrent processor cores are able to generate. At the same time, however, the experienced amount of interference increases the execution time.

Following this circular dependency, most processor-core-modular approaches to timing verification are iterative. We briefly describe this iterative nature for the case of a co-runner-sensitive WCET analysis (cf. Section 7.5). The principle behind most interference-aware schedulability analyses is similar [Schliecker and Ernst, 2010; Altmeyer et al., 2015]. Co-runner-sensitive WCET analysis typically starts by calculating an initial WCET bound under the optimistic assumption that the analyzed program does not experience any shared-resource interference. Subsequently, it calculates an upper bound on the amount of interference that the concurrent processor cores can generate in any time interval as long as the initial WCET bound. The resulting interference bound is used during the calculation of an updated WCET bound. The updated WCET bound is again used to calculate an updated bound on the amount of concurrent interference. This procedure is repeated until a fixed point is reached.

A function that maps a given length to an upper bound on the number of occurrences of a particular event that can happen on a system in any time interval of the given length is referred to as *arrival curve* [Boudec and Thiran, 2001]. Thus, the calculation of an upper bound on the number of occurrences of an interference-generating event of a processor core C_j that can happen in any time interval of a given length corresponds to the calculation of a value on an arrival curve. Such an interference-generating event can e.g. be the event that core C_j is granted access to a shared bus for one cycle.

The aforementioned iterative, processor-core-modular approaches to timing verification inherently rely on the calculation of values on arrival curves. In this chapter, we discuss different methods for calculating a value $\alpha_E(l)$ on an arrival curve α_E. Essentially all existing methods calculate curve values at the granularity of program runs (cf. Section 10.1). We present a method that calculates curve values at finer granularities and, thus, results in more precise curve values (cf. Section 10.2).

In our experiments, we calculate values on arrival curves for programs executed on a quad-core processor with a shared bus and a Round-Robin bus arbitration policy (cf. Figure 9.1). The considered processor cores shall feature an out-of-order pipeline, a local instruction cache of size 1KiB, and a local data cache of size 1KiB (i.e. processor core configuration $Conf_{ic}^{ooo}$ from Table 9.1).

In order to keep the software setup of the system under analysis simple, we assume that the considered processor core C_j repeatedly executes a single program in a non-preemptive fashion. Each discussed approach to the calculation of curve values is evaluated by calculating multiple arrival curve values $\alpha_E(l)$ for each benchmark listed in Table 9.2. For each benchmark, we calculate arrival curve values $\alpha_E(l)$ for each interval length $l \in \{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. In our experiments, we assume that E is the event occurring when the considered core C_j is granted access to the shared bus for one cycle ($Granted_{C_j}$, cf. Section 7.5). Note that all arrival curve values for a given benchmark and a given calculation method are calculated by the same instance of our prototype implementation. Thus, in particular, computationally complex preparation steps—as, e.g., the micro-architectural analysis and the graph construction—are only performed once per pair of benchmark and calculation method.

We conduct all experiments on a quad-core Intel® Core™ i7 processor clocked at 2.4 GHz and provided 16 GiB of main memory.

For the remainder of this chapter, we rely on the following naming conventions. The program $prog_{C_i}$ shall be the program which is executed on core C_i and for which we calculate a co-runner-sensitive WCET bound. The program $prog_{C_j}$ shall be the program which is executed on a concurrent core C_j of core C_i and for which we calculate arrival curve values during the co-runner-sensitive WCET analysis.

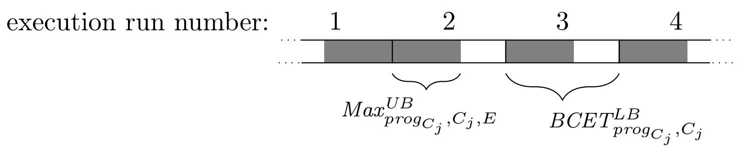


Figure 10.3.: Pathological worst-case scenario for calculating a value at a position l of an arrival curve. In the first considered execution run of $prog_{C_j}$, all occurrences of event E (marked in gray) are assumed to happen at the end. In all subsequent execution runs, all occurrences of event E are assumed to happen at the beginning.

In the following sections, we discuss the different methods for calculating a value $\alpha_E(l)$ on an arrival curve α_E.

10.1. Calculation at the Granularity of Program Runs

In this section, we discuss the calculation of values on arrival curves at the granularity of program runs. The resulting arrival curve is referred to as $\alpha_E^{prog\text{-}gran}$. Intuitively, this calculation pessimistically assumes that the occurrences of event E that an execution run of program $prog_{C_j}$ generates can be distributed across the execution run in an arbitrary manner. For simplicity, we only present this calculation in the context of the scenario that we consider in our experiments (i.e. non-preemptive scheduling, one program per processor core).

The presented calculation of values on arrival curves relies on existing event bounds that are sound with respect to every execution run of $prog_{C_j}$ on core C_j (cf. Chapter 6). $Max^{UB}_{prog_{C_j},C_j,E}$ shall be an upper bound on the number of occurrences of event E that can happen in any execution run of $prog_{C_j}$ on core C_j. $BCET^{LB}_{prog_{C_j},C_j}$ shall be a BCET bound for $prog_{C_j}$ on core C_j.

$$Max^{UB}_{prog_{C_j},C_j,E} \geq Maximum_{prog_{C_j},C_j,E} \tag{10.1}$$

$$BCET^{LB}_{prog_{C_j},C_j} \leq BCET_{prog_{C_j},C_j} \tag{10.2}$$

The value at position l of arrival curve $\alpha_E^{prog\text{-}gran}$ is calculated as the minimum of three bound values (cf. equation (10.3)). The three bound values are orthogonal in the sense that, for each of the bound values, there is an interval length l for which the bound value is more precise (i.e. strictly smaller) than the other two bound values.

$$\alpha_E^{prog\text{-}gran}(l) = \min\left(\alpha_{1,E}^{prog\text{-}gran}(l),\ \alpha_{2,E}^{prog\text{-}gran}(l),\ \alpha_{3,E}^{prog\text{-}gran}(l)\right) \tag{10.3}$$

The calculation of the three bound values relies on a pathological worst-case scenario in which every execution run of $prog_{C_j}$ takes $BCET^{LB}_{prog_{C_j},C_j}$ clock cycles and produces $Max^{UB}_{prog_{C_j},C_j,E}$ occurrences of event E. Moreover, it is pessimistically assumed that, in the first execution run of $prog_{C_j}$ which is spanned by the considered time interval of length l, all occurrences of event E happen at the end of the execution run. In each subsequent execution run of $prog_{C_j}$, all occurrences of event E shall happen at the beginning of the execution run. This *pathological worst-case scenario* is presented in Figure 10.3. Note that the time interval of length l is guaranteed to span the maximal number of occurrences of event E in this pathological scenario if it is aligned in a way that it begins with the events at the end of the first program run. This is demonstrated in Figure 10.4.

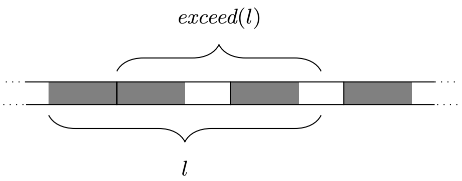


Figure 10.4.: In the pathological worst-case scenario, a time interval of length l spans the maximal number of occurrences of event E if it is aligned in a way that it begins with the events at the end of the first program run.

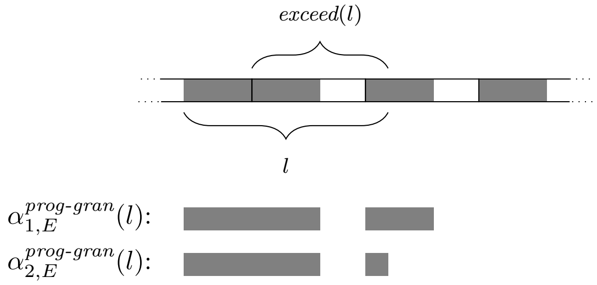

exceed(*l*)
l
$\alpha_{1,E}^{\textit{prog-gran}}(l)$:
$\alpha_{2,E}^{\textit{prog-gran}}(l)$:

Figure 10.5.: In case the considered time interval only spans some of the event occurrences of an execution run, bound value $\alpha_{1,E}^{\textit{prog-gran}}(l)$ is unnecessarily pessimistic.

The first bound value $\alpha_{1,E}^{\textit{prog-gran}}(l)$ always accounts for at least the $Max_{prog_{C_j},C_j,E}^{UB}$ occurrences of event E at the end of the first execution run. The helper function *exceed*(l) defines the length of the part of the considered time interval which does not overlap with the first execution run.

$$\textit{exceed}(l) = \max\left(0,\ l - Max_{prog_{C_j},C_j,E}^{UB}\right) \tag{10.4}$$

Bound value $\alpha_{1,E}^{\textit{prog-gran}}(l)$ accounts for the number of occurrences of event E that happen after the first execution run by pessimistically assuming that every started execution run contributes $Max_{prog_{C_j},C_j,E}^{UB}$ event occurrences.

$$\alpha_{1,E}^{\textit{prog-gran}}(l) = Max_{prog_{C_j},C_j,E}^{UB} + \left\lceil \frac{\textit{exceed}(l)}{BCET_{prog_{C_j},C_j}^{LB}} \right\rceil \cdot Max_{prog_{C_j},C_j,E}^{UB} \tag{10.5}$$

Note that bound value $\alpha_{1,E}^{\textit{prog-gran}}(l)$ is unnecessarily pessimistic in case the aligned time interval only spans some of the event occurrences of an execution run. This is demonstrated in Figure 10.5. Bound value $\alpha_{1,E}^{\textit{prog-gran}}(l)$ pessimistically assumes that all event occurrences of the third execution run are spanned by the time interval—which is not the case.

In order to avoid this pessimism, bound value $\alpha_{2,E}^{\textit{prog-gran}}(l)$ accounts for the number of occurrences of event E that happen after the first execution run by assuming that every fully covered execution run contributes $Max_{prog_{C_j},C_j,E}^{UB}$ event occurrences and every clock cycle of the remainder can contribute one event occurrence.

$$\begin{aligned}\alpha_{2,E}^{\textit{prog-gran}}(l) = Max_{prog_{C_j},C_j,E}^{UB} &+ \left\lfloor \frac{\textit{exceed}(l)}{BCET_{prog_{C_j},C_j}^{LB}} \right\rfloor \cdot Max_{prog_{C_j},C_j,E}^{UB} \\ &+ \left(\textit{exceed}(l) \bmod BCET_{prog_{C_j},C_j}^{LB}\right)\end{aligned} \tag{10.6}$$

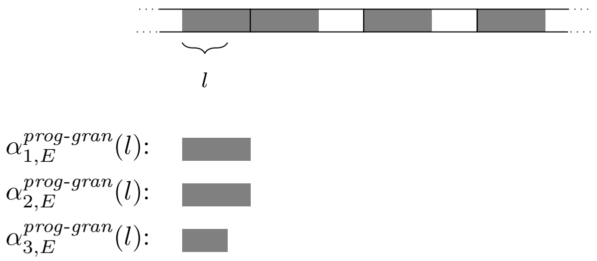

l
$\alpha^{prog\text{-}gran}_{1,E}(l)$:
$\alpha^{prog\text{-}gran}_{2,E}(l)$:
$\alpha^{prog\text{-}gran}_{3,E}(l)$:

Figure 10.6.: In case the length of the considered time interval is smaller than $Max^{UB}_{prog_{C_j},C_j,E}$, bound values $\alpha^{prog\text{-}gran}_{1,E}(l)$ and $\alpha^{prog\text{-}gran}_{2,E}(l)$ are unnecessarily pessimistic.

Consequently, bound value $\alpha^{prog\text{-}gran}_{2,E}(l)$ is more precise than bound value $\alpha^{prog\text{-}gran}_{1,E}(l)$ for the example in Figure 10.5. Note, however, that bound value $\alpha^{prog\text{-}gran}_{1,E}(l)$ is more precise than bound value $\alpha^{prog\text{-}gran}_{2,E}(l)$ for the example in Figure 10.4. This demonstrates that both bound values are orthogonal in the sense that none of them is for all cases at least as precise as the other.

Finally, bound value $\alpha^{prog\text{-}gran}_{3,E}(l)$ exploits that there can be at most l occurrences of event E in any time interval of at most l clock cycles.

$$\alpha^{prog\text{-}gran}_{3,E}(l) = l \tag{10.7}$$

Bound value $\alpha^{prog\text{-}gran}_{3,E}(l)$ is more precise than the previous two bound values in case the interval length l is strictly smaller than $Max^{UB}_{prog_{C_j},C_j,E}$. This is demonstrated in Figure 10.6. On the other hand, bound value $\alpha^{prog\text{-}gran}_{3,E}(l)$ is less precise than the previous two bound values for the examples in Figure 10.4 and Figure 10.5. Thus, all three bound values are orthogonal with respect to their precision.

Based on this intuitive consideration, we are confident that the calculated arrival curve value $\alpha_E(l)$ (i.e. the minimum of the three presented bound values) is sound with respect to the concrete traces. Moreover, we are confident that it is the most precise arrival curve value that can be calculated based on $BCET^{LB}_{prog_{C_j},C_j}$ and $Max^{UB}_{prog_{C_j},C_j,E}$ for the scenario that we consider. Formal proofs of these statements, however, are beyond the scope of this thesis.

In a more general execution scenario with preemptive scheduling, the calculation of values on arrival curves at the granularity of program runs does not rely on BCET bounds as multiple execution runs might be active at the same time on the same processor core. Instead, the calculation relies on upper bounds on the number of started execution runs in any time interval of a given length [Schliecker and Ernst, 2010]. Such upper bounds are typically calculated based on a minimum inter-arrival time of the considered program [Altmeyer et al., 2015]. Due to time and space constraints, however, we do not further discuss the calculation of values on arrival curves for such a more general scenario.

In an experiment, we evaluate the calculation of arrival curve values at the granularity of program runs for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The overall experiment takes one hour, 31 minutes, and 53.91 seconds—which is not significantly longer than the calculation of a co-runner-insensitive WCET bound per benchmark for the same hardware platform (cf. Figure B.5 in Appendix B). The additional experiment runtime of around 16 minutes is due to the curve value calculation requiring two event bounds per benchmark ($BCET^{LB}_{prog_{C_j},C_j}$ and $Max^{UB}_{prog_{C_j},C_j,E}$). The co-runner-insensitive WCET analysis, in contrast, only calculates a single event bound (namely the WCET bound) per benchmark. Thus, the amount of implicit path enumerations is doubled for the calculation of arrival curve values at the granularity of program runs. Due to its purely algebraic nature, the actual curve value calculation (cf. equation (10.3)) barely contributes to the experiment runtime.

Table 10.1 lists the resulting arrival curve values normalized to the respective interval lengths. The table demonstrates that the calculation at the granularity of program runs suffers from a poor precision. For each considered benchmark, every calculated arrival curve value is identical to the corresponding interval length. In the following, we describe—at an intuitive level—three fundamental drawbacks with respect to the precision of the presented calculation method.

The first drawback is the pessimistic assumption that any execution run of the considered program can take a minimal amount of clock cycles (namely $BCET^{LB}_{prog_{C_j},C_j}$) and—at the same time—can generate a maximal amount of occurrences of event E (namely $Max^{UB}_{prog_{C_j},C_j,E}$). For most real-world programs, however, the number of occurrences of the events that we consider (e.g. granted access cycles at the shared bus) is typically higher for longer execution runs. Thus, an execution run that only takes the BCET of the considered program is typically not able to generate the maximal amount of event occurrences (e.g. because there is no path through the control flow of the program that takes the minimal amount of clock cycles and—at the same time—generates the maximal number of occurrences of event E). As an extreme case, consider the case of the event bound $Max^{UB}_{prog_{C_j},C_j,E}$ being at least as high as the BCET bound $BCET^{LB}_{prog_{C_j},C_j}$. In this case, the calculation results in $\alpha_E(l) = l$ for each interval length l. Intuitively, the resulting identity arrival curve is of no use during co-runner-sensitive timing analysis as the analysis has to pessimistically assume that an interference-generating event can happen during every clock cycle. Thus, any obtained co-runner-sensitive WCET bound coincides with the corresponding co-runner-insensitive WCET bound. Note that, for programs with a wide range of possible execution times (i.e. there is a large gap between the BCET and the WCET), it is not unlikely that the exact upper bound on the number of event occurrences per program run ($Maximum_{prog_{C_j},C_j,E}$) is greater than the BCET ($BCET_{prog_{C_j},C_j}$) and, thus, also $Max^{UB}_{prog_{C_j},C_j,E}$ is greater than $BCET^{LB}_{prog_{C_j},C_j}$ according to equations (10.1) and (10.2).

But even if we assume the hypothetical case that program $prog_{C_j}$ only has a single concrete trace which takes BCET-many clock cycles and generates the maximal amount of occurrences of event E, the presented calculation method for values on arrival curves is still potentially overly pessimistic for relatively small interval lengths. Due to the considered pathological worst-case distribution of event occurrences (cf. Figure 10.3), the calculation cannot lead to a better result than $\alpha_E(l) = l$ in case $l \leq 2 \cdot Max^{UB}_{prog_{C_j},C_j,E}$. However, we suppose that, for most real-world programs, the occurrences of interference-generating events cannot be distributed across the execution runs as assumed by the pathological worst-case scenario. This pessimism is particularly problematic during co-runner-sensitive WCET analysis in case one of the interference-generating programs on the concurrent processor cores has a significantly longer execution time than the program for which a WCET bound is calculated. This is demonstrated in Figure 10.7. The co-runner-insensitive WCET bound of the program executed on core C_1 is smaller than twice the amount of interference an execution run of the program on core C_2 can generate. Thus, for any interval length l considered during co-runner-sensitive analysis, the corresponding arrival curve value $\alpha_E(l)$ is equal to l. As a consequence, the co-runner-sensitive analysis has to pessimistically assume that core C_2 generates enough interference to delay the program on core C_1 up to its co-runner-insensitive WCET bound.

The third drawback of the calculation of arrival curve values at the granularity of program runs is that this calculation inherently relies on the existence of a per-execution-run event bound $Max^{UB}_{prog_{C_j},C_j,E}$. For most diverging programs, however, there is no such per-execution-run event bound. Thus, in a scenario in which there are only strict timing requirements for the programs on some of the processor cores and there is a diverging program on one of the other processor cores, a co-runner-sensitive analysis calculating arrival curve values at the granularity of program runs is useless as it has to pessimistically assume that a time interval of any length can be filled up with occurrences of interference-generating events.

	10^4	10^5	10^6	10^7	10^8	10^9
cruise_control	1.000	1.000	1.000	1.000	1.000	1.000
digital_stopwatch	1.000	1.000	1.000	1.000	1.000	1.000
es_lift	1.000	1.000	1.000	1.000	1.000	1.000
flight_control	1.000	1.000	1.000	1.000	1.000	1.000
pilot	1.000	1.000	1.000	1.000	1.000	1.000
roboDog	1.000	1.000	1.000	1.000	1.000	1.000
trolleybus	1.000	1.000	1.000	1.000	1.000	1.000
lift	1.000	1.000	1.000	1.000	1.000	1.000
powerwindow	1.000	1.000	1.000	1.000	1.000	1.000
binarysearch	1.000	1.000	1.000	1.000	1.000	1.000
bsort	1.000	1.000	1.000	1.000	1.000	1.000
complex_updates	1.000	1.000	1.000	1.000	1.000	1.000
countnegative	1.000	1.000	1.000	1.000	1.000	1.000
fft	1.000	1.000	1.000	1.000	1.000	1.000
filterbank	1.000	1.000	1.000	1.000	1.000	1.000
fir2dim	1.000	1.000	1.000	1.000	1.000	1.000
iir	1.000	1.000	1.000	1.000	1.000	1.000
insertsort	1.000	1.000	1.000	1.000	1.000	1.000
jfdctint	1.000	1.000	1.000	1.000	1.000	1.000
lms	1.000	1.000	1.000	1.000	1.000	1.000
ludcmp	1.000	1.000	1.000	1.000	1.000	1.000
matrix1	1.000	1.000	1.000	1.000	1.000	1.000
md5	1.000	1.000	1.000	1.000	1.000	1.000
minver	1.000	1.000	1.000	1.000	1.000	1.000
pm	1.000	1.000	1.000	1.000	1.000	1.000
prime	1.000	1.000	1.000	1.000	1.000	1.000
sha	1.000	1.000	1.000	1.000	1.000	1.000
st	1.000	1.000	1.000	1.000	1.000	1.000
adpcm_dec	1.000	1.000	1.000	1.000	1.000	1.000
adpcm_enc	1.000	1.000	1.000	1.000	1.000	1.000
audiobeam	1.000	1.000	1.000	1.000	1.000	1.000
cjpeg_transupp	1.000	1.000	1.000	1.000	1.000	1.000
cjpeg_wrbmp	1.000	1.000	1.000	1.000	1.000	1.000
dijkstra	1.000	1.000	1.000	1.000	1.000	1.000
epic	1.000	1.000	1.000	1.000	1.000	1.000
g723_enc	1.000	1.000	1.000	1.000	1.000	1.000
gsm_dec	1.000	1.000	1.000	1.000	1.000	1.000
gsm_encode	1.000	1.000	1.000	1.000	1.000	1.000
h264_dec	1.000	1.000	1.000	1.000	1.000	1.000
huff_dec	1.000	1.000	1.000	1.000	1.000	1.000
mpeg2	1.000	1.000	1.000	1.000	1.000	1.000
ndes	1.000	1.000	1.000	1.000	1.000	1.000
petrinet	1.000	1.000	1.000	1.000	1.000	1.000
rijndael_dec	1.000	1.000	1.000	1.000	1.000	1.000
rijndael_enc	1.000	1.000	1.000	1.000	1.000	1.000
statemate	1.000	1.000	1.000	1.000	1.000	1.000
susan	1.000	1.000	1.000	1.000	1.000	1.000
—**average**—	1.000	1.000	1.000	1.000	1.000	1.000
			1.000			

Table 10.1.: Arrival curve values calculated at the granularity of program runs for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The presented values are normalized to the respective interval lengths.

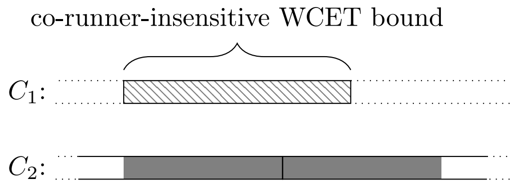


Figure 10.7.: In this example, the co-runner-insensitive WCET bound of the program on core C_1 is smaller than twice the amount of interference an execution run of the program on core C_2 can generate. Due to the pathological distribution of the interference on core C_2 that is assumed by the curve value calculation (cf. Figure 10.3), a co-runner-sensitive analysis calculating arrival curve values at the granularity of program runs cannot result in a WCET bound that is more precise than the co-runner-insensitive one.

In order to overcome the described drawbacks, in the next section, we present the calculation of arrival curve values at significantly finer granularities (e.g. at the granularity of basic blocks, cf. Section 6.4.4).

10.2. Calculation at Finer Granularities

To the best of our knowledge, the calculation of arrival curve values at granularities finer than program granularity has so far only been proposed for an execution model in which each program is a sequence of subprograms—so called superblocks [Pellizzoni et al., 2010]. For each superblock, there are upper and lower bounds on the execution times of the superblock as well as an upper bound on the numbers of occurrences of interference events during the execution of the superblock. This superblock model is universal in the sense that every program can naturally be represented by a single superblock, which is the program itself. If a program is represented by a single superblock, the event bounds for this superblock can be obtained as presented in Chapter 6. If every program is represented by a single superblock, however, the corresponding calculation of arrival curve values effectively also only operates at the granularity of program runs and, thus, suffers from the fundamental drawbacks described in Section 10.1. On the other hand, it is so far completely unclear how to represent a general program (featuring non-trivial control flow, cf. Section 6.4.1) by a fine-grained sequence of superblocks. This means that the calculation of arrival curve values at granularities significantly finer than program granularity is so far limited to relatively restricted classes of programs (e.g. single-path programs [Puschner, 2003]).

In this section, we present a calculation of arrival curve values that is—in principle—applicable to arbitrary programs. It argues about all subpaths of a fine-grained graph representation. The presented calculation can e.g. be applied to graphs at the granularity of cycle transitions (cf. Section 6.4.2) or to graphs at the granularity of basic blocks (cf. Section 6.4.4). In order to efficiently argue about all subpaths of the considered graph, the presented calculation resorts to implicit path enumeration (cf. Section 6.5). For the sake of readability, we split up the presentation of the proposed calculation into multiple subsections focusing on different aspects of the calculation.

10.2.1. Calculation Based on the Subpaths of a Graph

The presented calculation of arrival curve values is based on a graph G^C as it is used for the calculation of per-execution-run event bounds in Section 6.4. As we base the presented calculation on all subpaths anyway, it is not necessary to add a set of feedback nodes of the graph to its end nodes.

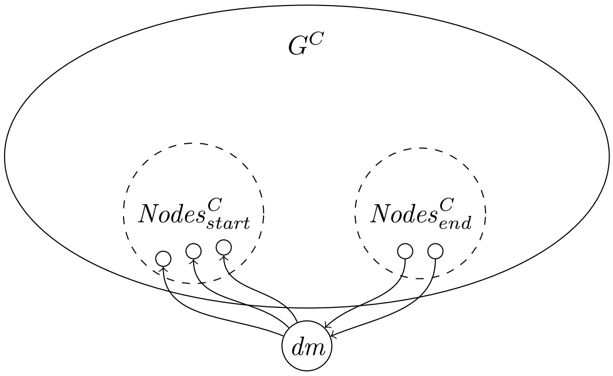


Figure 10.8.: The time interval that is considered during the calculation of an arrival curve value may span across multiple execution runs of the analyzed program. Thus, the presented calculation is based on a graph G^D in which a new execution run starts once an execution run ends. To this end, graph G^D introduces a fresh dummy node *dm* which has zero-weighted outgoing edges to all start nodes of G^C and zero-weighted incoming edges from all end nodes of G^C.

In contrast to the calculation of per-execution-run event bounds, the time interval of length l that is considered during the calculation of an arrival curve value $\alpha_E(l)$ may span across multiple execution runs of the analyzed program. Thus, the calculation is based on a modified version G^D of graph G^C in which a new execution run starts once an execution run ends. To this end, graph G^D introduces a fresh dummy node *dm* which has zero-weighted outgoing edges to all start nodes of G^C and zero-weighted incoming edges from all end nodes of G^C. This principle is depicted in Figure 10.8. The fresh dummy node *dm* shall be the single start node and the single end node of graph G^D. Note that this graph transformation is equivalent to the transformation that is formalized in Section 9.4.

The actual calculation of arrival curve value $\alpha_E(l)$ is based on all subpaths of graph G^D. In case the original graph G^C is *at the granularity of cycle transitions* (cf. Section 6.4.2), the calculation is completely straight forward. It considers all subpaths of graph G^D for which the sum of the lower bounds on the number of clock cycles along the subpath does not exceed the interval length l. In the following, this inequation is referred to as *window constraint*. The resulting set of considered subpaths is referred to as $\widehat{SubPaths}^D_{\leq l}$.

$$WindowConstr(\widehat{p}, l) \Leftrightarrow \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCycle}^{LB}(\widehat{p}, x) \leq l \tag{10.8}$$

$$\widehat{SubPaths}^D_{\leq l} = \left\{ \widehat{p} \in \widehat{SubPaths}^D \;\middle|\; WindowConstr(\widehat{p}, l) \right\} \tag{10.9}$$

An arrival curve value $Bound^{subp}_{1,E}(l)$ is obtained by maximizing the sum of the upper bounds on the number of occurrences of event E along the subpath over all considered subpaths.

$$Obj^{subp}_{1,E}(\widehat{p}) = \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wE}^{UB}(\widehat{p}, x) \tag{10.10}$$

$$Bound^{subp}_{1,E}(l) = \max_{\widehat{p} \in \widehat{SubPaths}^D_{\leq l}} Obj^{subp}_{1,E}(\widehat{p}) \tag{10.11}$$

In case the original graph G^C is *not at the granularity of cycle transitions* (i.e. there is an edge that argues about a sequence of multiple cycle transitions, e.g. about a sequence of cycle transitions through a basic block, cf. Section 6.4.4), the window constraint has to be slightly adapted. In a similar way as in Section 10.1, we assume a pathological worst-case scenario (cf. Figure 10.3): For the first edge of a considered subpath, all occurrences of event E shall happen at the end of the edge. For all subsequent edges of a considered subpath, all occurrences of event E shall happen in the beginning of the edge. Note that the time interval of length l is guaranteed to span the maximal number of occurrences of event E in this pathological scenario if it is aligned in a way that it begins with the events at the end of the first edge (cf. Figure 10.4). We only consider subpaths for which a time interval of length l aligned in such a way fully covers all edges of the subpath except the first and the last edge. This is reflected by the following alternative definition of the window constraint.

$$\mathit{WindowConstr}(\widehat{p}, l) \Leftrightarrow \sum_{x \in \{0\} \cap \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wE^{UB}}(\widehat{p}, x) + \sum_{x \in \mathbb{N}_{>0} \cap \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wCycle^{LB}}(\widehat{p}, x) \leq l \tag{10.12}$$

The arrival curve value $\mathit{Bound}^{subp}_{1,E}(l)$ is also sound in case the original graph G^C is not at the granularity of cycle transitions. In this case, however, it is not necessarily the most precise arrival curve value that we can calculate based on subpaths of the graph. The following objective function $\mathit{Obj}^{subp}_{2,E}(\widehat{p}, l)$ is orthogonal to $\mathit{Obj}^{subp}_{1,E}(\widehat{p})$ in the sense that, for certain subpaths $\widehat{p}$, it results in strictly more precise values. Intuitively, $\mathit{Obj}^{subp}_{2,E}(\widehat{p}, l)$ omits the upper bound on the number of occurrences of event E for the last edge in the subpath. Instead, it adds the amount by which the interval length l exceeds the left-hand side of the window constraint. The idea behind this alternative objective function is very similar to the idea behind curve value $\alpha^{prog\text{-}gran}_{2,E}(l)$ presented in Section 10.1.

$$\begin{aligned}
\mathit{Obj}^{subp}_{2,E}(\widehat{p}, l) &= \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wE^{UB}}(\widehat{p}, x) \\
&\quad + l - \sum_{x \in \{0\} \cap \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wE^{UB}}(\widehat{p}, x) - \sum_{x \in \mathbb{N}_{>0} \cap \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wCycle^{LB}}(\widehat{p}, x) \\
&= \sum_{x \in \mathbb{N}_{>0} \cap \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wE^{UB}}(\widehat{p}, x) + l - \sum_{x \in \mathbb{N}_{>0} \cap \mathbb{N}_{<\mathit{len}(\widehat{p})-1}} \widehat{wCycle^{LB}}(\widehat{p}, x)
\end{aligned} \tag{10.13}$$

Note that, for any subpath $\widehat{p}$ with $\mathit{len}(\widehat{p}) \in \{0, 1\}$ and any interval length l, the value of the alternative objective function is guaranteed to coincide with the interval length l. Thus, there is no sense in calculating an arrival curve value $\mathit{Bound}^{subp}_{2,E}(l)$ directly based on the alternative objective function $\mathit{Obj}^{subp}_{2,E}(\widehat{p}, l)$ as such a curve value would always coincide with the interval length l. Instead, we calculate a combined arrival curve value $\mathit{Bound}^{subp}_{comb,E}(l)$ by taking the minimum of both objective values for each subpath.

$$\mathit{Bound}^{subp}_{comb,E}(l) = \max_{\widehat{p} \in \widehat{\mathit{SubPaths}}^{D}_{\leq l}} \min \left\{ \mathit{Obj}^{subp}_{1,E}(\widehat{p}),\ \mathit{Obj}^{subp}_{2,E}(\widehat{p}, l) \right\} \tag{10.14}$$

We expect that the explicit enumeration of the members of $\widehat{\mathit{SubPaths}}^{D}_{\leq l}$ leads to an intractably high computational complexity. Thus, we do not experimentally evaluate the calculation of arrival curve values based on this set. Instead, in Section 10.2.3 and Section 10.2.4, we demonstrate and evaluate an implicit enumeration of the members of $\widehat{\mathit{SubPaths}}^{D}_{\leq l}$.

So far, we only described the calculation of values on upper-bounding arrival curves (i.e. upper-bounding the number of occurrences of event E within any time interval of at most l cycle transitions). Analogously, one can use the subpaths of a graph for the calculation of values on lower-bounding arrival curves (i.e. lower-bounding the number of occurrences of event E within any time interval of at least l cycle transitions). Such lower-bounding arrival curves are also referred to as *service curves* in the context of the network calculus [Boudec and Thiran, 2001]. Moreover, one can generalize this principle to the calculation of values on arrival curves that bound the number of occurrences of one event in dependence on the number of occurrences of another event, e.g. upper-bounding (lower-bounding) the number of cache misses within any sequence of at most (least) l cache accesses. A detailed discussion of the calculation of values on lower-bounding and/or generalized arrival curves, however, is beyond the scope of this thesis. Nonetheless, note that slight variants of the implicit enumeration of subpaths presented in Section 10.2.3 and Section 10.2.4 are applicable to these calculation scenarios as well.

10.2.2. Using Lifted System Properties to Detect Infeasible Subpaths

As for the calculation of per-execution-run event bounds (cf. Section 6.4.3), we can use lifted versions of system properties in order to improve the precision of the calculated arrival curve values. In this context, the lifted system properties are used to prune subpaths of the graph that are guaranteed to not describe any concrete traces.

$$Less\widehat{SubPaths}^{D}_{\leq l} = \left\{ \widehat{p} \in \widehat{SubPaths}^{D}_{\leq l} \;\middle|\; \forall P_k \in Prop_{prog_{C_i},C_i} : \widehat{P^{path}_k}(\widehat{p}) \right\} \tag{10.15}$$

Recall, however, that the overall goal of co-runner-sensitive WCET analysis (cf. Section 7.5) is to calculate a WCET bound for program $prog_{C_i}$ executed on core C_i. Thus, the corresponding set $ExecRuns_{prog_{C_i},C_i}$ contains all concrete traces that are exhibited during an execution run of program $prog_{C_i}$ on core C_i. Consequently, for the calculation of arrival curve values during this co-runner-sensitive analysis, we use the set $Prop_{prog_{C_i},C_i}$ of system properties that hold during every execution run of program $prog_{C_i}$ on core C_i.

The arrival curve values calculated in our co-runner-sensitive analysis bound the number of occurrences of events which are determined by the operation of the programs executed on the concurrent cores C_j of core C_i (i.e. $C_j \neq C_i$, cf. Algorithm 7.1 and Algorithm 7.2). Thus, a calculation of a value on an arrival curve overapproximates the events that occur on core C_j while core C_i executes a run of program $prog_{C_i}$. In particular, this means that the considered program execution sequences of core C_j could start in the middle of a program execution run and end in the middle of a (possibly different) program execution run.

Classically, WCET analysis only uses system properties that are valid for a single execution run of a program on the modeled processor core or a prefix of such an execution run (i.e. those system properties of set $Prop_{prog_{C_i},C_i}$ that argue about the events occurring on core C_i for which a WCET bound is calculated). The calculation of values on arrival curves, in contrast, relies on system properties that are valid even if the concrete trace starts or ends at an arbitrary program location on the modeled processor core C_j ($\neq C_i$).

We demonstrate this difference for the case of the well-known control flow property of a *relative loop bound* [Li and Malik, 1995]. Figure 10.9 shows an excerpt of an example program. The excerpt is chosen in a way that it only contains the basic blocks of a loop (bb_1 and bb_2) as well as the single predecessor basic block of the loop (bb_0). In this thesis, we only consider reducible loops, which have a unique loop header (here bb_1). For simplicity, the body of our example loop also only consists of a single basic block (bb_2). For the example loop, there shall be a relative upper loop bound $loop_1^{UB}$. This means that, every time the loop is entered (i.e. the loop header is entered from outside of the loop, in this case from bb_0), the loop body (here bb_2) may at most be executed $loop_1^{UB}$ times.

If we perform a WCET analysis of the example program (i.e. the example program is executed on core C_i for which we calculate a WCET bound), we consider all control flow paths from the start of the program. As the program cannot start from inside of the loop, the loop can only be entered via basic block bb_0. Consequently, in any control flow path from the program

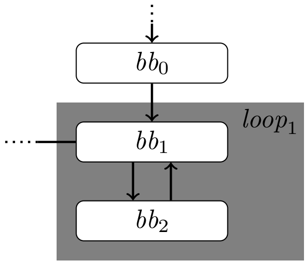


Figure 10.9.: Excerpt of an example program. It shows the basic blocks of a loop (bb_1 and bb_2) as well as the single predecessor basic block of the loop (bb_0).

start, the number of executions of the loop body may at most be $loop_1^{UB}$ times as high as the number of executions of basic block bb_0. This relative upper loop bound is reflected by the following control flow property (cf. Section 6.4.5), which is typically used during the calculation of per-execution-run event bounds (cf. Chapter 6).

$$\begin{aligned} P_{loop_1^{UB}}(\mathit{flow}) \Leftrightarrow & \left| \left\{ \ x \in \mathbb{N}_{<len(\mathit{flow})} \ \middle| \ \mathit{flow}(x) = \mathit{front}(bb_2) \ \right\} \right| \\ & \leq loop_1^{UB} \cdot \left| \left\{ \ x \in \mathbb{N}_{<len(\mathit{flow})} \ \middle| \ \mathit{flow}(x) = \mathit{front}(bb_0) \ \right\} \right| \end{aligned} \tag{10.16}$$

This control flow property can be lifted to the level of approximation of paths through a graph as follows. The lifted version fulfills the soundness criterion presented in Section 6.4.5. A formal proof of this, however, is omitted due to time and space constraints.

$$\widehat{P^{path}_{loop_1^{UB}}}(\widehat{p}) \Leftrightarrow \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCF^{LB}_{front(bb_2)}}(\widehat{p}, x) \leq loop_1^{UB} \cdot \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCF^{UB}_{front(bb_0)}}(\widehat{p}, x) \tag{10.17}$$

If, in contrast, the example program is executed on core C_j for which we calculate arrival curve values, we also have to consider concrete traces for which the operation of core C_j starts inside of the example loop (i.e. in basic block bb_1 or bb_2). Such concrete traces may perform up to $loop_1^{UB}$ more executions of the loop body than $loop_1^{UB}$ times the number of executions of basic block bb_0. This is reflected by the following control flow property, which can be used during the calculation of values on arrival curves.

$$\begin{aligned} P_{loop_1^{UB}}(\mathit{flow}) \Leftrightarrow & \left| \left\{ \ x \in \mathbb{N}_{<len(\mathit{flow})} \ \middle| \ \mathit{flow}(x) = \mathit{front}(bb_2) \ \right\} \right| \\ & \leq loop_1^{UB} \cdot \left(\left| \left\{ \ x \in \mathbb{N}_{<len(\mathit{flow})} \ \middle| \ \mathit{flow}(x) = \mathit{front}(bb_0) \ \right\} \right| + 1 \right) \end{aligned} \tag{10.18}$$

This control flow property can be lifted to the level of approximation of subpaths of a graph as follows.

$$\widehat{P^{path}_{loop_1^{UB}}}(\widehat{p}) \Leftrightarrow \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCF^{LB}_{front(bb_2)}}(\widehat{p}, x) \leq loop_1^{UB} \cdot \left(\sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCF^{UB}_{front(bb_0)}}(\widehat{p}, x) + 1 \right) \tag{10.19}$$

Note that it is overly pessimistic to always account for the additional $loop_1^{UB}$ loop iterations as proposed by equation (10.18). Instead, it suffices to account for the additional loop iterations only in case the considered control flow starts inside of the loop. This is reflected by the following improved version of the control flow property of equation (10.18).

$$\begin{aligned} P_{loop_1^{UB}}(flow) \Leftrightarrow & \left| \left\{ \ x \in \mathbb{N}_{<len(flow)} \ \middle| \ flow(x) = front(bb_2) \ \right\} \right| \\ & \leq loop_1^{UB} \cdot \left(\left| \left\{ \ x \in \mathbb{N}_{<len(flow)} \ \middle| \ flow(x) = front(bb_0) \ \right\} \right| \right. \\ & \left. + \left| \left\{ \ x \in \mathbb{N}_{<len(flow)} \cap \{0\} \ \middle| \ flow(x) \in \{front(bb_1), front(bb_2)\} \ \right\} \right| \right) \end{aligned} \tag{10.20}$$

This improved version of the control flow property can be lifted to the level of approximation of subpaths of a graph as follows.

$$\begin{aligned} \widehat{P^{path}_{loop_1^{UB}}}(\widehat{p}) \Leftrightarrow \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCF^{LB}_{front(bb_2)}}(\widehat{p}, x) \leq loop_1^{UB} \cdot & \left(\sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{wCF^{UB}_{front(bb_0)}}(\widehat{p}, x) \right. \\ & \left. + \left| \left\{ \ x \in \mathbb{N}_{<len(\widehat{p})} \cap \{0\} \ \middle| \ (\widehat{p}(x), \widehat{p}(x+1)) \in insideEdges^D(loop_1) \ \right\} \right| \right) \end{aligned} \tag{10.21}$$

The set $insideEdges^D(loop_1)$ is a subset of $Edges^D$ that has to contain every start edge of every subpath of graph G^D that describes a control flow in which a basic block of the loop is at the front position. A more detailed discussion of this criterion is beyond the scope of this thesis. A formalization of this criterion would require a formalization of control flows that can span across multiple execution runs (we presented a similar formalization of the control flows of an execution run in Section 6.4.1 and Section 6.4.5). Such a formalization is omitted due to time and space constraints.

Note that, in case multiple instances of the same loop can be nested due to recursion, the corresponding loop bound property for the calculation of arrival curve values has to also take into account an upper bound on the number of nested instances of the loop. Intuitively, in such a case, the considered subpath could start in the first iteration of the innermost loop instance and every outer instance could also be in its first iteration. The benchmarks that we consider during our experiments (cf. Table 10.10), however, do not make use of recursion.

Further note that, in a similar way as for the loop bound properties, *cache persistence properties* also differ between the calculation of per-execution-run event bounds and the calculation of arrival curve values. For any prefix of a single execution run, the number of cache misses of a persistent memory block is upper-bounded by the number of times the corresponding persistence scope is entered. During the calculation of arrival curve values, however, we also have to take into account control flow on the considered core which starts in the middle of a persistence scope. Our experiments, however, do not take into account cache persistence.

10.2.3. Implicit Enumeration of the Subpaths

As a preparation for the implicit enumeration of the subpaths of graph G^D, we derive graph G^F, which marks all nodes of graph G^D as start nodes and end nodes. Note that we skipped the letter E in this naming scheme, as it already stands for the event E.

$$\begin{aligned} G^F &= (\mathit{Nodes}^F, \mathit{Nodes}^F_{start}, \mathit{Nodes}^F_{end}, \mathit{Edges}^F) && (10.22)\\ \mathit{Nodes}^F &= \mathit{Nodes}^D && (10.23)\\ \mathit{Nodes}^F_{start} &= \mathit{Nodes}^D && (10.24)\\ \mathit{Nodes}^F_{end} &= \mathit{Nodes}^D && (10.25)\\ \mathit{Edges}^F &= \mathit{Edges}^D && (10.26) \end{aligned}$$

Consequently, the set of paths through graph G^F coincides with the set of subpaths of graph G^D.

$$\widehat{\mathit{Paths}}^F = \widehat{\mathit{SubPaths}}^D \tag{10.27}$$

Thus, an implicit enumeration of the paths through graph G^F corresponds to an implicit enumeration of the subpaths of graph G^D. We exploit this in order to implicitly enumerate the members of $\widehat{\mathit{SubPaths}}^D_{\leq l}$. In case the original graph G^C is at the granularity of cycle transitions (cf. equation (10.9)), the implicit enumeration is performed as follows.

$$\mathit{WindowConstr}((tt, is, ie), l) \Leftrightarrow \sum_{edg \in \mathit{Edges}^F} tt(edg) \cdot w\widehat{\mathit{Cycle}}^{LB}(edg) \leq l \tag{10.28}$$

$$\widehat{\mathit{Implicit}}^F_{\leq l} = \left\{ (tt, is, ie) \in \widehat{\mathit{Implicit}}^F \;\middle|\; \mathit{WindowConstr}((tt, is, ie), l) \right\} \tag{10.29}$$

In case the original graph G^C is not at the granularity of cycle transitions (cf. equation (10.12)), the window constraint has to be slightly adapted.

$$\begin{aligned} \mathit{WindowConstr}((tt, is, ie), l) \Leftrightarrow & \sum_{edg \in \mathit{Edges}^F} [tt(edg) - is(edg) - ie(edg)] \cdot w\widehat{\mathit{Cycle}}^{LB}(edg) \\ & + \sum_{edg \in \mathit{Edges}^F} is(edg) \cdot \widehat{wE}^{UB}(edg) \leq l \end{aligned} \tag{10.30}$$

As a next step, we use lifted versions of system properties in order to prune infeasible implicit paths from the set $\widehat{\mathit{Implicit}}^F_{\leq l}$.

$$\mathit{Less}\widehat{\mathit{Implicit}}^F_{\leq l} = \left\{ (tt, is, ie) \in \widehat{\mathit{Implicit}}^F_{\leq l} \;\middle|\; \forall P_k \in \mathit{Prop}_{\mathit{prog}_{C_i}, C_i} : \widehat{P}_k^{impli}((tt, is, ie)) \right\} \tag{10.31}$$

The improved version of the relative-loop-bound system property on subpaths of the graph (cf. equation (10.21)) is lifted to the level of approximation of the implicit enumeration of subpaths as follows.

$$\begin{aligned} \widehat{P}^{impli}_{loop_1^{UB}}((tt, is, ie)) \Leftrightarrow & \sum_{edg \in \mathit{Edges}^F} tt(edg) \cdot w\widehat{CF}^{LB}_{front(bb_2)}(edg) \\ & \leq loop_1^{UB} \cdot \Big(\sum_{edg \in \mathit{Edges}^F} tt(edg) \cdot w\widehat{CF}^{UB}_{front(bb_0)}(edg) \\ & \qquad + \sum_{edg \in \mathit{insideEdges}^D(loop_1)} is(edg) \Big) \end{aligned} \tag{10.32}$$

The arrival curve value $Bound^{subp}_{1,E}(l)$ (cf. equation (10.11)) is safely overapproximated based on the implicit enumeration as follows.

$$Obj^{impli}_{1,E}((tt, is, ie)) = \sum_{edg \in Edges^F} tt(edg) \cdot \widehat{wE^{UB}}(edg) \tag{10.33}$$

$$Bound^{impli}_{1,E}(l) = \max_{(tt,is,ie) \in \widehat{LessImplicit}^F_{\leq l}} Obj^{impli}_{1,E}((tt, is, ie)) \tag{10.34}$$

The combined arrival curve value $Bound^{subp}_{comb,E}(l)$ (cf. equation (10.14)) is safely overapproximated based on the implicit enumeration as follows.

$$\begin{aligned} Obj^{impli}_{2,E}((tt, is, ie), l) = & \sum_{edg \in Edges^F} [tt(edg) - is(edg) - ie(edg)] \cdot \widehat{wE^{UB}}(edg) \\ & + l - \sum_{edg \in Edges^F} [tt(edg) - is(edg) - ie(edg)] \cdot \widehat{wCycle}^{LB}(edg) \end{aligned} \tag{10.35}$$

$$Bound^{impli}_{comb,E}(l) = \max_{(tt,is,ie) \in \widehat{LessImplicit}^F_{\leq l}} \min \left\{ Obj^{impli}_{1,E}((tt, is, ie)),\ Obj^{impli}_{2,E}((tt, is, ie), l) \right\} \tag{10.36}$$

As the ILP solver that we use[1] does not directly support combining two objectives with a minimum operation, we simulate the minimum by an auxiliary variable *obj* that is upper-bounded by each argument of the minimum operation.

$$Bound^{impli}_{comb,E}(l) = \max_{(tt,is,ie,obj) \in \widehat{LessImplicit}^F_{\leq l} \times \mathbb{N}} obj \tag{10.37}$$

$$obj \leq Obj^{impli}_{1,E}((tt, is, ie)) \tag{10.38}$$

$$obj \leq Obj^{impli}_{2,E}((tt, is, ie), l) \tag{10.39}$$

In case one of the calculated arrival curve values exceeds the length l of the considered time interval, it is reset to l (i.e. within any time interval of l clock cycles, there cannot be more than l occurrences of event E).

$$Bound^{impli}_{1,E}(l) \longleftarrow \min \left\{ l,\ Bound^{impli}_{1,E}(l) \right\} \tag{10.40}$$

$$Bound^{impli}_{comb,E}(l) \longleftarrow \min \left\{ l,\ Bound^{impli}_{comb,E}(l) \right\} \tag{10.41}$$

Note that some equations of the presented calculations rely on the information whether a given edge is the start edge or the end edge of the current subpath (i.e. they argue about the components *is* and *ie* of the implicit subpath). Thus, it is not possible to trivially lift these calculations to an implicit enumeration without binary variables (cf. Section 9.4). For a formal sketch on how to safely lift these calculations to an implicit enumeration without binary variables, we refer to Section 10.2.4. In this subsection, we conduct an experimental evaluation of the implicit enumeration with binary variables.

In our experiments, we perform the implicit enumeration of subpaths on graphs that are fully node-sensitive at basic block boundaries, node-insensitive inside of basic blocks, and fully edge-weight-sensitive with respect to the upper bound on the number of occurrences of event E (cf. Section 6.4.4). As the implicit enumeration of subpaths tends to be significantly more complex than the calculation of per-execution-run event bounds (cf. Chapter 6), we enforce a time

[1] http://www.gurobi.com/products/gurobi-optimizer

limit of ten minutes per calculation of an arrival curve value. In case the ILP solver performing the implicit enumeration is not finished after this time limit, we use the best upper bound that the solver has calculated so far.

We evaluate the calculation of arrival curve values by implicit subpath enumeration for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. We start with an experimental evaluation of the implicit subpath enumeration calculating $Bound_{1,E}^{impli}(l)$. The experiment takes 13 hours, 14 minutes, and 50.43 seconds. On average, the calculation of arrival curve values for a benchmark takes 7.65 times as long as the corresponding calculations at the granularity of program runs (cf. Figure B.22 on page 339 of Appendix B). The memory consumption compared to the corresponding calculations at the granularity of program runs increases by an average factor of 4.30 (cf. Figure B.23 on page 340 of Appendix B). Table 10.2 lists the resulting arrival curve values normalized to the respective interval lengths. For each benchmark, there are six normalized curve values in Table 10.2. Starting from an interval length of 10^5, the normalized curve values of a benchmark typically tend toward a constant factor less than one (representing the worst-case event density of the benchmark). Intuitively, as soon as the interval length exceeds the length of the hottest path through the benchmark, the hottest path is likely to be executed repeatedly until the interval length is filled up. The benchmarks `epic`, `mpeg2`, and `susan` are exceptions: some of their curve value calculations hit the time limit of ten minutes before there is an upper bound on the curve value that is more precise than the interval length. On average, the calculated curve values are 0.891 times as high as the corresponding interval lengths. The average factor of 0.891 is calculated as the geometric mean of all normalized arrival curve values. This average factor corresponds to an average reduction of 10.9 percent compared to the calculation at the granularity of program runs (cf. Table 10.1). Note that the arrival curve value $Bound_{1,E}^{impli}(l)$ is up to 41.8 percent reduced compared to the calculation at the granularity of program runs (e.g. for benchmark `st` in combination with interval length 10^7, cf. Table 10.1).

The experimental evaluation of the implicit subpath enumeration calculating $Bound_{comb,E}^{impli}(l)$ takes 15 hours, 32 minutes, and 37.34 seconds. On average, the calculation of arrival curve values for a benchmark takes 9.98 times as long as the corresponding calculations at the granularity of program runs (cf. Figure B.24 on page 341 of Appendix B). The memory consumption compared to the corresponding calculations at the granularity of program runs increases by an average factor of 4.35 (cf. Figure B.25 on page 342 of Appendix B). This means that, in terms of runtime and memory consumption, the implicit subpath enumeration calculating $Bound_{comb,E}^{impli}(l)$ is more demanding than the implicit subpath enumeration calculating $Bound_{1,E}^{impli}(l)$ (cf. Figure B.22 on page 339 and Figure B.23 on page 340 of Appendix B).

Table 10.3 lists the calculated arrival curve values $Bound_{comb,E}^{impli}(l)$ normalized to the respective interval lengths l. On average, the calculated curve values are 0.893 times as high as the corresponding interval lengths. Thus, the increased computational complexity and memory consumption of the implicit subpath enumeration calculating $Bound_{comb,E}^{impli}(l)$ compared to the implicit subpath enumeration calculating $Bound_{1,E}^{impli}(l)$ is not justified by an improved precision (cf. Table 10.2).

Note that the normalized curve values for $Bound_{comb,E}^{impli}(l)$ are at best slightly more precise than the normalized curve values for $Bound_{1,E}^{impli}(l)$ (up to 0.001 as e.g. for benchmark `petrinet` in combination with interval length 10^4). The potential precision improvement of $Bound_{comb,E}^{impli}(l)$ compared to $Bound_{1,E}^{impli}(l)$ is limited to a more precise bounding of the number of occurrences of event E for the start edge and the end edge of the considered implicit subpaths. We follow that, for reasonable interval lengths (i.e. $l \geq 10^4$) and graphs at the granularity of basic blocks, the precision improvement normalized to the respective interval length is negligible.

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	0.927	0.924	0.923	0.923	0.923	0.923
`digital_stopwatch`	0.951	0.942	0.941	0.941	0.941	0.941
`es_lift`	0.959	0.957	0.957	0.957	0.957	0.957
`flight_control`	0.955	0.915	0.909	0.908	0.908	0.908
`pilot`	0.858	0.848	0.848	0.847	0.847	0.847
`roboDog`	0.952	0.950	0.949	0.949	0.949	0.949
`trolleybus`	0.966	0.962	0.961	0.961	0.961	0.961
`lift`	0.827	0.812	0.810	0.810	0.810	0.810
`powerwindow`	0.942	0.937	0.937	0.933	0.927	0.927
`binarysearch`	0.922	0.920	0.920	0.920	0.920	0.920
`bsort`	0.912	0.911	0.910	0.910	0.910	0.910
`complex_updates`	0.868	0.866	0.866	0.866	0.866	0.866
`countnegative`	0.832	0.829	0.829	0.829	0.829	0.829
`fft`	0.899	0.896	0.858	0.853	0.852	0.852
`filterbank`	0.840	0.786	0.778	0.774	0.773	0.773
`fir2dim`	0.837	0.800	0.797	0.797	0.797	0.797
`iir`	0.879	0.877	0.876	0.876	0.876	0.876
`insertsort`	0.860	0.853	0.852	0.852	0.852	0.852
`jfdctint`	0.836	0.822	0.821	0.820	0.820	0.820
`lms`	0.871	0.867	0.866	0.866	0.866	0.866
`ludcmp`	0.952	0.950	0.950	0.950	0.950	0.950
`matrix1`	0.796	0.791	0.791	0.791	0.791	0.791
`md5`	0.900	0.888	0.887	0.887	0.887	0.887
`minver`	0.880	0.872	0.871	0.870	0.870	0.870
`pm`	0.938	0.917	0.906	0.904	0.904	0.904
`prime`	0.906	0.903	0.903	0.903	0.903	0.903
`sha`	0.901	0.851	0.845	0.845	0.845	0.845
`st`	0.669	0.624	0.587	0.582	0.582	0.582
`adpcm_dec`	0.861	0.852	0.851	0.851	0.851	0.851
`adpcm_enc`	0.853	0.841	0.840	0.840	0.840	0.840
`audiobeam`	0.934	0.931	0.930	0.930	0.930	0.930
`cjpeg_transupp`	0.946	0.943	0.942	0.942	0.942	0.942
`cjpeg_wrbmp`	0.910	0.908	0.908	0.908	0.908	0.908
`dijkstra`	0.922	0.919	0.918	0.918	0.918	0.918
`epic`	1.000	1.000	1.000	1.000	1.000	0.947
`g723_enc`	0.940	0.937	0.936	0.936	0.936	0.936
`gsm_dec`	0.941	0.936	0.935	0.935	0.935	0.935
`gsm_encode`	0.858	0.796	0.782	0.779	0.778	0.778
`h264_dec`	0.929	0.913	0.911	0.911	0.911	0.911
`huff_dec`	0.906	0.904	0.904	0.904	0.904	0.904
`mpeg2`	1.000	1.000	1.000	1.000	1.000	1.000
`ndes`	0.961	0.893	0.884	0.883	0.883	0.883
`petrinet`	0.934	0.931	0.930	0.930	0.930	0.930
`rijndael_dec`	0.987	0.981	0.980	0.980	0.980	0.980
`rijndael_enc`	0.988	0.979	0.978	0.978	0.978	0.978
`statemate`	0.947	0.941	0.941	0.941	0.941	0.941
`susan`	1.000	1.000	1.000	1.000	1.000	1.000
—**average**—	0.905	0.892	0.889	0.888	0.888	0.887
			0.891			

Table 10.2.: Arrival curve values calculated by implicit subpath enumeration $(Bound^{impli}_{1,E}(l))$ for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The presented values are normalized to the respective interval lengths.

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	0.927	0.924	0.923	0.923	0.923	0.923
`digital_stopwatch`	0.951	0.942	0.941	0.941	0.941	0.941
`es_lift`	0.959	0.957	0.957	0.957	0.957	0.957
`flight_control`	0.955	0.915	0.909	0.908	0.908	0.908
`pilot`	0.858	0.848	0.848	0.847	0.847	0.847
`roboDog`	0.952	0.950	0.949	0.949	0.949	0.949
`trolleybus`	0.966	0.962	0.961	0.961	0.961	0.961
`lift`	0.827	0.812	0.810	0.810	0.810	0.810
`powerwindow`	0.941	0.937	0.937	0.933	0.927	0.927
`binarysearch`	0.921	0.920	0.920	0.920	0.920	0.920
`bsort`	0.912	0.911	0.910	0.910	0.910	0.910
`complex_updates`	0.868	0.866	0.866	0.866	0.866	0.866
`countnegative`	0.832	0.829	0.829	0.829	0.829	0.829
`fft`	0.899	0.896	0.857	0.853	0.852	0.852
`filterbank`	0.840	0.786	0.778	0.774	0.773	0.773
`fir2dim`	0.837	0.800	0.797	0.797	0.797	0.797
`iir`	0.878	0.877	0.876	0.876	0.876	0.876
`insertsort`	0.860	0.853	0.852	0.852	0.852	0.852
`jfdctint`	0.836	0.822	0.821	0.820	0.820	0.820
`lms`	0.870	0.867	0.866	0.866	0.866	0.866
`ludcmp`	0.951	0.950	0.950	0.950	0.950	0.950
`matrix1`	0.796	0.791	0.791	0.791	0.791	0.791
`md5`	0.900	0.888	0.887	0.887	0.887	0.887
`minver`	0.880	0.871	0.871	0.870	0.870	0.870
`pm`	1.000	0.917	0.906	1.000	1.000	1.000
`prime`	0.906	0.903	0.903	0.903	0.903	0.903
`sha`	0.901	0.851	0.845	0.845	0.845	0.845
`st`	0.669	0.624	0.587	0.582	0.582	0.582
`adpcm_dec`	0.860	0.852	0.851	0.851	0.851	0.851
`adpcm_enc`	0.853	0.841	0.840	0.840	0.840	0.840
`audiobeam`	0.933	0.931	0.930	0.930	0.930	0.930
`cjpeg_transupp`	0.946	0.943	0.942	0.942	0.942	1.000
`cjpeg_wrbmp`	0.910	0.908	0.908	0.908	0.908	0.908
`dijkstra`	0.921	0.919	0.918	0.918	0.918	0.918
`epic`	1.000	1.000	1.000	1.000	1.000	1.000
`g723_enc`	0.939	0.937	0.936	0.936	0.936	0.936
`gsm_dec`	0.940	0.936	0.935	0.935	0.935	0.935
`gsm_encode`	0.862	0.796	0.782	0.779	0.778	0.778
`h264_dec`	0.929	0.913	0.911	0.911	0.911	0.911
`huff_dec`	0.905	0.904	0.904	0.904	0.904	0.904
`mpeg2`	1.000	1.000	1.000	1.000	1.000	1.000
`ndes`	0.961	0.893	0.884	0.883	0.883	0.883
`petrinet`	0.933	0.930	0.930	0.930	0.930	0.930
`rijndael_dec`	0.990	0.981	0.980	0.980	0.980	0.980
`rijndael_enc`	0.989	0.979	0.978	0.978	0.978	0.978
`statemate`	0.947	0.941	0.941	0.941	0.941	0.941
`susan`	1.000	1.000	1.000	1.000	1.000	1.000
—**average**—	0.906	0.892	0.889	0.890	0.890	0.891
			0.893			

Table 10.3.: Arrival curve values calculated by implicit subpath enumeration ($Bound^{impli}_{comb,E}(l)$) for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The presented values are normalized to the respective interval lengths.

Further note that, in some cases, the normalized curve values for $Bound^{impli}_{comb,E}(l)$ are greater (i.e. less precise) than the normalized curve values for $Bound^{impli}_{1,E}(l)$ (up to 0.096 as e.g. for benchmark `pm` in combination with interval length 10^9). Based solely on the formal specifications of $Bound^{impli}_{1,E}(l)$ and $Bound^{impli}_{comb,E}(l)$ (cf. equations (10.34) and (10.36)), this reduction of precision cannot be explained. The potential reduction of precision in our experiments is due to the enforced time limit of ten minutes per implicit subpath enumeration: Since the computational complexity of calculating $Bound^{impli}_{comb,E}(l)$ is significantly higher than the computational complexity of calculating $Bound^{impli}_{1,E}(l)$, the best upper bound on $Bound^{impli}_{comb,E}(l)$ that is reached after ten minutes is in some cases greater than the best upper bound on $Bound^{impli}_{1,E}(l)$ that is reached after ten minutes.

We conclude that the precision of $Bound^{impli}_{1,E}(l)$ and $Bound^{impli}_{comb,E}(l)$ is significantly higher than the precision of curve values calculated at the granularity of program runs. The precision of $Bound^{impli}_{comb,E}(l)$ is on a par with the precision of $Bound^{impli}_{1,E}(l)$. The computational complexity and memory consumption of calculating $Bound^{impli}_{comb,E}(l)$, however, are significantly higher than the computational complexity and memory consumption of calculating $Bound^{impli}_{1,E}(l)$. Consequently, in the remainder of this chapter, implicit subpath enumeration with binary variables is only used for the calculation of $Bound^{impli}_{1,E}(l)$.

In the next subsection, we experimentally evaluate whether an overapproximation of $Bound^{impli}_{1,E}(l)$ at the level of implicit subpath enumeration without binary variables (cf. Section 9.4) is beneficial in terms of computational complexity and/or memory consumption.

10.2.4. Implicit Subpath Enumeration without Binary Variables

The implicit subpath enumeration without binary variables is performed on a graph G^G that is obtained from graph G^F as described in Section 9.4. In case the original graph G^C is at the granularity of cycle transitions (cf. equation (10.9)), the implicit enumeration without binary variables considers the following set $\widehat{Implicit}^{G\prime}_{\leq l}$, which is a trivial overapproximation of $\widehat{Implicit}^{F}_{\leq l}$ (cf. equation (10.29)).

$$WindowConstr(tt, l) \Leftrightarrow \sum_{edg \in Edges^G} tt(edg) \cdot \widehat{wCycle}^{LB}(edg) \leq l \tag{10.42}$$

$$\widehat{Implicit}^{G\prime}_{\leq l} = \left\{ \; tt \in \widehat{Implicit}^{G\prime} \;\middle|\; WindowConstr(tt, l) \; \right\} \tag{10.43}$$

In case the original graph G^C is not at the granularity of cycle transitions (cf. equation (10.12)), the implicit enumeration without binary variables considers the following adapted window constraint. The resulting set $\widehat{Implicit}^{G\prime}_{\leq l}$ is still an overapproximation of $\widehat{Implicit}^{F}_{\leq l}$, but no longer a trivial one. A soundness proof of this statement is omitted due to time and space constraints.

$$\begin{aligned} WindowConstr(tt, l) \Leftrightarrow & \sum_{edg \in Edges^G} tt(edg) \cdot \widehat{wCycle}^{LB}(edg) \\ & - \sum_{(*, nd) \in Edges^G_{start}} tt((*, nd)) \\ & \qquad \cdot \left[\max_{edg \in outEdges(nd)} \left(\widehat{wCycle}^{LB}(edg) - \widehat{wE}^{UB}(edg) \right) \right] \\ & - \sum_{(nd, *) \in Edges^G_{end}} tt((nd, *)) \cdot \left[\max_{edg \in inEdges(nd)} \widehat{wCycle}^{LB}(edg) \right] \leq l \end{aligned} \tag{10.44}$$

As a next step, we use lifted versions of system properties in order to prune infeasible implicit paths from the set $\widehat{Implicit}^{G'}_{\leq l}$.

$$\widehat{LessImplicit}^{G'}_{\leq l} = \left\{ \, tt \in \widehat{Implicit}^{G'}_{\leq l} \;\middle|\; \forall P_k \in Prop_{prog_{C_i}, C_i} : \widehat{P^{impli'}_{k}}(tt) \, \right\} \tag{10.45}$$

The improved version of the relative-loop-bound system property on implicit subpaths with binary variables (cf. equation (10.32)) can be lifted to the level of approximation of implicit subpaths without binary variables as follows.

$$\begin{aligned} \widehat{P^{impli'}_{loop_1^{UB}}}(tt) \Leftrightarrow & \sum_{edg \in Edges^G} tt(edg) \cdot \widehat{wCF^{LB}_{front(bb_2)}}(edg) \\ & \leq loop_1^{UB} \cdot \Bigg(\sum_{edg \in Edges^G} tt(edg) \cdot \widehat{wCF^{UB}_{front(bb_0)}}(edg) \\ & \quad + \sum_{edg \in \{(*, nd) \in Edges^G_{start} \mid outEdges(nd) \cap insideEdges^D(loop_1) \neq \emptyset\}} tt(edg) \Bigg) \end{aligned} \tag{10.46}$$

The arrival curve value $Bound^{impli}_{1,E}(l)$ (cf. equation (10.34)) is trivially overapproximated based on the implicit enumeration without binary variables as follows.

$$Obj^{impli'}_{1,E}(tt) = \sum_{edg \in Edges^G} tt(edg) \cdot \widehat{wE^{UB}}(edg) \tag{10.47}$$

$$Bound^{impli'}_{1,E}(l) = \max_{tt \in \widehat{LessImplicit}^{G'}_{\leq l}} Obj^{impli'}_{1,E}(tt) \tag{10.48}$$

The combined arrival curve value $Bound^{impli}_{comb,E}(l)$ (cf. equation (10.36)) is safely overapproximated based on the implicit enumeration without binary variables as follows.

$$\begin{aligned} Obj^{impli'}_{2,E}(tt, l) = & \sum_{edg \in Edges^G} tt(edg) \cdot \Big[\, \widehat{wE^{UB}}(edg) - \widehat{wCycle^{LB}}(edg) \, \Big] \\ & + \sum_{(*, nd) \in Edges^G_{start}} tt((*, nd)) \\ & \qquad \cdot \Big[\max_{edg \in outEdges(nd)} \big(\widehat{wCycle^{LB}}(edg) - \widehat{wE^{UB}}(edg) \big) \Big] \\ & + \sum_{(nd, *) \in Edges^G_{end}} tt((nd, *)) \\ & \qquad \cdot \Big[\max_{edg \in inEdges(nd)} \big(\widehat{wCycle^{LB}}(edg) - \widehat{wE^{UB}}(edg) \big) \Big] \\ & + l \end{aligned} \tag{10.49}$$

$$Bound^{impli'}_{comb,E}(l) = \max_{tt \in \widehat{LessImplicit}^{G'}_{\leq l}} \min \left\{ \, Obj^{impli'}_{1,E}(tt), \; Obj^{impli'}_{2,E}(tt, l) \, \right\} \tag{10.50}$$

As the ILP solver that we use[2] does not directly support combining two objectives with a minimum operation, we simulate the minimum by an auxiliary variable obj that is upper-bounded by each argument of the minimum operation.

$$Bound_{comb,E}^{impli\prime}(l) = \max_{(tt,obj) \in Less\widehat{Implicit}_{\leq l}^{G\prime} \times \mathbb{N}} obj \tag{10.51}$$

$$obj \leq Obj_{1,E}^{impli\prime}(tt) \tag{10.52}$$

$$obj \leq Obj_{2,E}^{impli\prime}(tt, l) \tag{10.53}$$

In case one of the calculated arrival curve values exceeds the length l of the considered time interval, it is reset to l (i.e. within any time interval of l clock cycles, there cannot be more than l occurrences of event E).

$$Bound_{1,E}^{impli\prime}(l) \longleftarrow \min \left\{ \; l, \; Bound_{1,E}^{impli\prime}(l) \; \right\} \tag{10.54}$$

$$Bound_{comb,E}^{impli\prime}(l) \longleftarrow \min \left\{ \; l, \; Bound_{comb,E}^{impli\prime}(l) \; \right\} \tag{10.55}$$

In our experiments, we perform the implicit subpath enumeration without binary variables on graphs that are fully node-sensitive at basic block boundaries, node-insensitive inside of basic blocks, and fully edge-weight-sensitive with respect to the upper bound on the number of occurrences of event E (cf. Section 6.4.4). In the same way as in Section 10.2.3, we enforce a time limit of ten minutes per calculation of an arrival curve value. In case the ILP solver performing the implicit enumeration is not finished after this time limit, we use the best upper bound that the solver has calculated so far.

Table 10.4 lists the resulting arrival curve values normalized to the respective interval lengths. The normalized curve values for $Bound_{1,E}^{impli\prime}(l)$ are up to 0.053 greater (for benchmark `st` in combination with interval length 10^4) and up to 0.053 smaller (e.g. for benchmark `epic` in combination with interval length 10^7) than the normalized curve values for $Bound_{1,E}^{impli}(l)$ (cf. Table 10.2). The normalized curve values for $Bound_{1,E}^{impli\prime}(l)$ have an average value (geometric mean) of 0.893. This means that the precision of $Bound_{1,E}^{impli\prime}(l)$ is on average almost on a par with the precision of $Bound_{1,E}^{impli}(l)$ (0.891, cf. Table 10.2).

The overall experiment for $Bound_{1,E}^{impli\prime}(l)$ takes 13 hours, five minutes, and 37.3 seconds (cf. Figure B.26 on page 343 of Appendix B) and, thus, around as long as the corresponding experiment for $Bound_{1,E}^{impli}(l)$ (cf. Figure B.22 on page 339 of Appendix B). Nonetheless, the average runtime increase of 8.82 compared to a curve value calculation at the granularity of program runs is slightly higher than the corresponding factor of 7.65 for implicit subpath enumeration with binary variables. The average increase in memory consumption of 3.04 (cf. Figure B.27 on page 344 of Appendix B), in contrast, is lower than the average increase in memory consumption for implicit subpath enumeration with binary variables (4.30, cf. Figure B.23 on page 340 of Appendix B). Taking a closer look at the non-normalized runtime and memory consumption of the experiments with $Bound_{1,E}^{impli}(l)$ and $Bound_{1,E}^{impli\prime}(l)$, we see that it significantly depends on the actual benchmark whether implicit subpath enumeration with or without binary variables is more beneficial in terms of computational complexity or memory consumption. Thus, based on these experimental results, we cannot recommend one of both approaches to implicit subpath enumeration over the other.

For simplicity and due to time and space constraints, in the remainder of this chapter, we only resort to implicit subpath enumeration with binary variables. Note, however, that all further principles introduced in the remainder of this chapter could as well be realized using implicit subpath enumeration without binary variables.

[2] http://www.gurobi.com/products/gurobi-optimizer

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	0.932	0.924	0.923	0.923	0.923	0.923
`digital_stopwatch`	0.966	0.944	0.942	0.941	0.941	0.941
`es_lift`	0.964	0.958	0.957	0.957	0.957	0.957
`flight_control`	0.955	0.936	0.911	0.909	0.908	0.908
`pilot`	0.877	0.851	0.849	0.849	0.848	0.847
`roboDog`	0.958	0.950	0.950	0.949	0.949	0.949
`trolleybus`	0.976	0.963	0.961	0.961	0.961	0.961
`lift`	0.842	0.814	0.810	0.810	0.810	0.810
`powerwindow`	0.947	0.938	0.937	0.937	0.928	0.927
`binarysearch`	0.922	0.920	0.920	0.920	0.920	0.920
`bsort`	0.912	0.911	0.910	0.910	0.910	0.910
`complex_updates`	0.870	0.866	0.866	0.866	0.866	0.866
`countnegative`	0.835	0.830	0.829	0.829	0.829	0.829
`fft`	0.906	0.897	0.886	0.857	0.853	0.852
`filterbank`	0.846	0.808	0.785	0.776	0.774	0.773
`fir2dim`	0.844	0.811	0.799	0.798	0.797	0.797
`iir`	0.880	0.877	0.876	0.876	0.876	0.876
`insertsort`	0.865	0.854	0.852	0.852	0.852	0.852
`jfdctint`	0.870	0.825	0.821	0.820	0.820	0.820
`lms`	0.881	0.868	0.866	0.866	0.866	0.866
`ludcmp`	0.954	0.950	0.950	0.950	0.950	0.950
`matrix1`	0.799	0.792	0.791	0.791	0.791	0.791
`md5`	0.901	0.888	0.887	0.887	0.887	0.887
`minver`	0.912	0.878	0.873	0.872	0.871	0.871
`pm`	0.942	0.936	0.918	0.906	0.904	0.904
`prime`	0.907	0.903	0.903	0.903	0.903	0.903
`sha`	0.918	0.865	0.847	0.845	0.845	0.845
`st`	0.722	0.634	0.604	0.583	0.582	0.582
`adpcm_dec`	0.881	0.855	0.851	0.851	0.851	0.851
`adpcm_enc`	0.861	0.845	0.842	0.842	0.841	0.840
`audiobeam`	0.935	0.931	0.931	0.930	0.930	0.930
`cjpeg_transupp`	0.951	0.943	0.942	0.942	0.942	0.942
`cjpeg_wrbmp`	0.913	0.908	0.908	0.908	0.908	0.908
`dijkstra`	0.925	0.919	0.919	0.918	0.918	0.918
`epic`	0.958	0.949	0.947	0.947	0.947	1.000
`g723_enc`	0.940	0.937	0.936	0.936	0.936	0.936
`gsm_dec`	0.945	0.936	0.935	0.935	0.935	0.935
`gsm_encode`	0.886	0.796	0.785	0.779	0.778	0.778
`h264_dec`	0.941	0.914	0.912	0.911	0.911	0.911
`huff_dec`	0.906	0.904	0.904	0.904	0.904	0.904
`mpeg2`	0.973	0.958	0.951	0.950	1.000	1.000
`ndes`	0.965	0.958	0.891	0.884	0.883	0.883
`petrinet`	0.939	0.931	0.930	0.930	0.930	0.930
`rijndael_dec`	1.000	0.984	0.981	0.980	0.980	0.980
`rijndael_enc`	1.000	0.983	0.978	0.978	0.978	0.978
`statemate`	0.959	0.943	0.941	0.941	0.941	0.941
`susan`	1.000	1.000	1.000	1.000	1.000	1.000
—**average**—	0.913	0.895	0.889	0.886	0.887	0.888
			0.893			

Table 10.4.: Arrival curve values calculated by implicit subpath enumeration without binary variables ($Bound_{1,E}^{impli'}(l)$) for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The presented values are normalized to the respective interval lengths.

Finally, we point out that the normalized curve values obtained from implicit subpath enumeration (with or without binary variables) are still relatively high. On average, almost 90 percent of a considered interval length can be filled up with bus access cycles (cf. Table 10.2 and Table 10.4). In Section 10.5, we demonstrate that a co-runner-sensitive WCET analysis relying on such high arrival curve values is typically not able to calculate more precise WCET bounds than a co-runner-insensitive WCET analysis. In order to reduce the relative amount of shared-bus interference that a processor core can generate, in Section 10.3, we propose to enforce minimum inter-start times during program scheduling.

10.2.5. Supporting Multiple Programs on the Same Processor Core

In this thesis, for simplicity, we only evaluate the calculation of values on arrival curves for a scenario in which every processor core executes a single program in a non-preemptive fashion. Nonetheless, the presented technique (i.e. the implicit enumeration of subpaths of a graph representation) also naturally applies to a scenario in which every processor core executes multiple programs in a non-preemptive fashion. In Figure 10.8, we demonstrate the construction of graph G^D by adding a dummy node dm to graph G^C and connecting the dummy node to the start nodes and end nodes of graph G^C. In the more general case of multiple programs $prog_{C_j}$ being executed on the same processor core C_j, we have one graph $G^{C,prog_{C_j}}$ per program $prog_{C_j}$ executed on the considered processor core. In this more general case, we obtain graph G^D by connecting all graphs $G^{C,prog_{C_j}}$ via the dummy node dm. Thus, the dummy node dm has zero-weighted outgoing edges to every start node of every graph $G^{C,prog_{C_j}}$ and zero-weighted incoming edges from every end node of every graph $G^{C,prog_{C_j}}$. This principle is demonstrated in Figure 10.10 for a scenario in which the programs $prog_1$ and $prog_2$ are executed on the considered processor core.

Additionally, it is possible to hard-wire limited knowledge about the program schedule directly into graph G^D. Such limited knowledge might e.g. be a static order in which the programs are executed. Or it might only be known that a given program is never executed twice in sequence without executing a different program in between. Figure 10.11 demonstrates the incorporation of such knowledge for the example in Figure 10.10. In this case, there shall be a static program order in which the programs $prog_1$ and $prog_2$ take turns at executing. In Figure 10.11, graph G^D expresses this knowledge by making use of two dummy nodes. Conceptually, the graph in Figure 10.11 can be seen as the result of applying a sequence of two graph transformations (cf. Section 6.4.6) to the graph in Figure 10.10. First, the graph is blown up (cf. equation (6.C12)) by creating specialized dummy nodes representing the case that $prog_1$ just terminated (dm_1) and the case that $prog_2$ just terminated (dm_2). Subsequently, infeasible edges (cf. equation (6.C13)) of the blown-up graph are removed, e.g. the outgoing edges of dummy node dm_1 targeting a start node of graph $G^{C,prog_1}$.

Note, however, that we expect the computational complexity of this approach to be unmanageable when dealing with multiple real-world programs executed on the same processor core. Moreover, even if the computational complexity is still bearable, we expect this approach to not be well-suited for a scenario with continuous development: In case only one of the programs of a processor core changes during development, the very costly calculations of arrival curve values (incorporating all programs of the core) have to be redone from scratch during the timing verification. To overcome these drawbacks, Section 10.4 sketches a more modular calculation of arrival curve values: The computationally complex aspects are calculated at a per-program level and the actual calculation of arrival curve values only uses the results of the per-program calculations. Yet, the sketched approach takes into account which part of a program can produce which amount of event occurrences and, thus, avoids the pessimism of a calculation at the granularity of program runs (cf. Section 10.1).

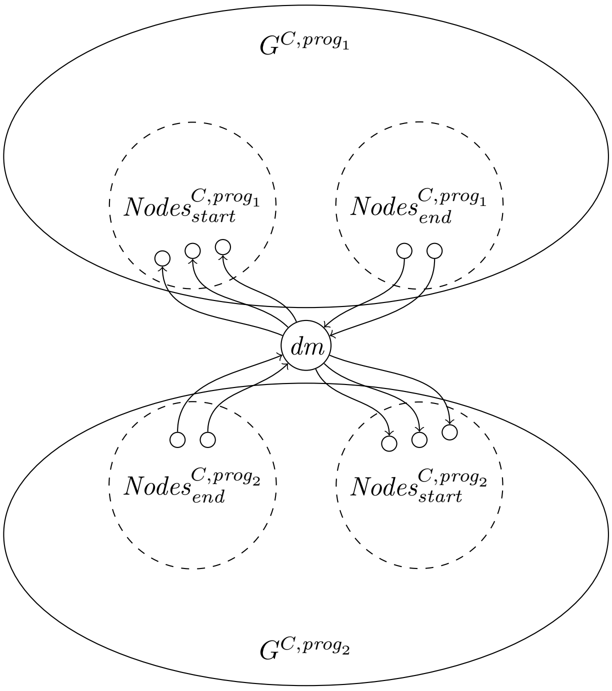


Figure 10.10.: In a scenario in which multiple programs are executed on the considered processor core, the calculation of arrival curve values is performed on a graph G^D that is obtained by connecting the graphs of all executed programs. To this end, graph G^D introduces a fresh dummy node *dm* which has zero-weighted outgoing edges to all start nodes of every per-program graph and zero-weighted incoming edges from all end nodes of every per-program graph.

10.2.6. Toward Preemptive Scheduling

Note that, in a setting with preemptive scheduling, preemption effects must be accounted for as they can change the operation of preempted programs (in particular the temporal distribution of event occurrences). Researchers have studied the safe modeling of the impact that preemptions have on the contents of caches [Altmeyer, 2013]. However, it is unclear how to safely and efficiently model the impact of preemptions on the operation of other components (e.g. pipelines) of real-world processors.

Even under the hypothetical assumption that there was a micro-architectural analysis that safely takes into account all possible preemption effects for a given real-world hardware platform, the presented calculation of arrival curve values based on subpaths of a graph would also have to be adapted in order to be sound for systems with preemptive scheduling. So far, the presented calculation inherently assumes that a context switch in the middle of an execution run is not possible (cf. Figure 10.8 and Figure 10.10). We expect that an adaption of the calculation for systems with preemptive scheduling would necessarily lead to the introduction of many additional graph edges accounting for the possible context switches. A more detailed sketch of such an adaption, however, exceeds the scope of this thesis.

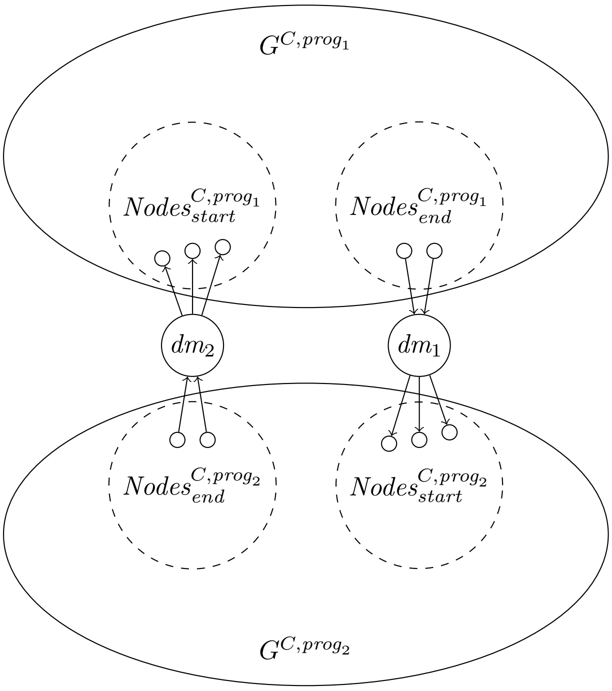


Figure 10.11.: Limited scheduling knowledge can optionally be hard-wired into graph G^D in order to improve the precision of the calculated arrival curve values. This is demonstrated for the example in Figure 10.10: The additional knowledge that both programs take turns at executing can be expressed in the graph by using two dummy nodes.

10.3. Enforcing Minimum inter-Start Times of Programs to Adjust the Amount of Generated Interference

In order to potentially reduce the number of occurrences of event E that a processor core can generate in a time interval of a given length, we would like to configure the minimum amount of time that an execution run of a program has to take. To this end, the non-preemptive program scheduler of the concrete system shall enforce a *minimum inter-start time* $MIST_{prog}$ per program *prog*. This means that the scheduler shall guarantee that, when an execution run of *prog* on a given core starts, a potential subsequent execution run of any program on the same core has to start at least $MIST_{prog}$ clock cycles later. The following equation formalizes this requirement.

$$\begin{aligned}
&\forall prog_1, prog_2 \in Programs : \forall C_i, C_j \in Cores : \\
&\quad \forall t \in ExecRuns_{prog_1,C_i} : \\
&\qquad \forall x \in \mathbb{N}_{<len(t)} : \\
&\qquad\quad Start_{prog_2,C_j}(t,x) \\
&\qquad\quad \Rightarrow \forall y \in \mathbb{N}_{>x} \cap \mathbb{N}_{<x+MIST_{prog_2}} \cap \mathbb{N}_{<len(t)} : \\
&\qquad\qquad\quad \neg\exists prog_3 \in Programs : Start_{prog_3,C_j}(t,y)
\end{aligned} \tag{10.56}$$

The minimum inter-start times of the programs executed on a processor core can be used to *adjust the density of occurrences of an event E* that is generated by the programs executed on a considered core. In the context of preemptive scheduling, the assumption of a minimum inter-release time per task [Altmeyer et al., 2015] respectively an arrival curve for releases per task [Schliecker and Ernst, 2010] has an analogous impact on the density of event occurrences.

To the best of our knowledge, the impact of incorporating such density-adjusting assumptions compared to pessimistically assuming a fully utilized system (i.e. assuming that the scheduler permanently executes programs on the considered core) has so far not yet been evaluated.

In this section, we demonstrate how to take into account the minimum inter-start times during the calculation of arrival curve values in order to potentially improve the precision of the calculated values. To this end, we extend the calculation methods presented in Section 10.1 and Section 10.2.

Note that a minimum inter-start time of a program that does not exceed the BCET of the program has no effect on the concrete traces of the concrete system (i.e. any execution run is at least as long as the minimum inter-start time anyway). Further note that, in case the minimum inter-start time of a program exceeds the WCET bound considered for the program during schedulability analysis, the minimum inter-start time of the program has to additionally be taken into account during schedulability analysis (e.g. by pessimistically using the minimum inter-start time instead of the WCET bound).

We expect that it is in general very challenging to individually choose the minimum inter-start times of all programs in a way that all timing requirements of the system are fulfilled. A more detailed discussion of the corresponding *optimization problem*, however, is beyond the scope of this thesis. Instead, in order to keep the setup of the experiments in this section simple, we choose a fixed minimum inter-start time per program. It shall be defined by a WCET bound assuming the absence of interference plus 50 percent of the additional execution time that the co-runner-insensitive WCET bound assumes. We refer to this as a *relative minimum inter-start time of* 0.5 ($relMIST_{0.5}$).

$$MIST_{prog_{C_j}} = WCET^{UB,0\text{-}interfer}_{prog_{C_j},C_j} + 0.5 \cdot (\,WCET^{UB,insens}_{prog_{C_j},C_j} - WCET^{UB,0\text{-}interfer}_{prog_{C_j},C_j}) \qquad (10.57)$$

10.3.1. Incorporation during Calculation at the Granularity of Program Runs

The calculation of arrival curve values at the granularity of program runs (cf. Section 10.1) is extended to take into account the minimum inter-start time by replacing all occurrences of the BCET bound $BCET^{LB}_{prog_{C_j},C_j}$ by the maximum of the BCET bound $BCET^{LB}_{prog_{C_j},C_j}$ and the minimum inter-start time $MIST_{prog_{C_j}}$. Thus, the extension fully preserves the simple and algebraic nature of the presented calculation at the granularity of program runs.

The experimental evaluation of the calculation of arrival curve values at the granularity of program runs incorporating a minimum inter-start time is conducted in the same way as the experiments of the preceding sections. Table 10.5 lists the resulting arrival curve values normalized to the respective interval lengths. The normalized curve values have an average value (geometric mean) of 0.594. This means that, on average, around sixty percent of a time interval can be filled up with bus access cycles. This result demonstrates that enforcing a minimum inter-start time can effectively reduce the amount of shared-bus interference that a processor core can generate according to our processor-core-modular modeling scheme as, without enforcing a minimum inter-start time, we are at best able to obtain an average value of 0.891 (cf. Table 10.2).

The overall experiment takes one hour, 36 minutes, and 34.69 seconds (cf. Figure B.28 on page 345 of Appendix B). This is relatively close to the overall runtime of the experiment calculating arrival curve values at the granularity of program runs without enforcing a minimum inter-start time (one hour, 31 minutes, and 53.91 seconds), which is expected as the incorporation of the minimum inter-start time preserves the simple and algebraic nature of the calculation at the granularity of program runs. In the same way, the memory consumption also only slightly increases compared to a calculation at the granularity of program runs without enforcing a minimum inter-start time (average increase factor of 1.01, cf. Figure B.29 on page 346 of Appendix B).

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	1.000	1.000	0.452	0.369	0.362	0.361
`digital_stopwatch`	1.000	1.000	0.797	0.374	0.343	0.338
`es_lift`	1.000	1.000	0.413	0.372	0.364	0.364
`flight_control`	1.000	1.000	1.000	0.412	0.364	0.356
`pilot`	1.000	1.000	0.461	0.363	0.351	0.350
`roboDog`	1.000	1.000	0.494	0.387	0.373	0.372
`trolleybus`	1.000	1.000	0.934	0.417	0.364	0.360
`lift`	1.000	1.000	1.000	0.672	0.370	0.350
`powerwindow`	1.000	1.000	1.000	1.000	0.464	0.377
`binarysearch`	0.405	0.366	0.362	0.361	0.361	0.361
`bsort`	1.000	1.000	1.000	0.429	0.357	0.352
`complex_updates`	0.554	0.388	0.363	0.360	0.360	0.359
`countnegative`	1.000	0.486	0.388	0.369	0.368	0.368
`fft`	1.000	1.000	1.000	1.000	1.000	0.758
`filterbank`	1.000	1.000	1.000	1.000	0.442	0.336
`fir2dim`	1.000	0.511	0.358	0.349	0.348	0.347
`iir`	0.429	0.372	0.368	0.368	0.368	0.368
`insertsort`	1.000	0.443	0.367	0.356	0.355	0.355
`jfdctint`	1.000	0.387	0.348	0.342	0.342	0.342
`lms`	1.000	1.000	0.937	0.375	0.342	0.338
`ludcmp`	1.000	0.603	0.362	0.350	0.348	0.348
`matrix1`	1.000	1.000	0.413	0.353	0.348	0.347
`md5`	1.000	1.000	1.000	1.000	1.000	0.440
`minver`	1.000	0.512	0.359	0.347	0.346	0.346
`pm`	1.000	1.000	1.000	1.000	0.534	0.348
`prime`	1.000	0.441	0.264	0.248	0.247	0.247
`sha`	1.000	1.000	1.000	1.000	0.467	0.359
`st`	1.000	1.000	0.511	0.323	0.306	0.304
`adpcm_dec`	1.000	0.447	0.365	0.355	0.355	0.354
`adpcm_enc`	1.000	0.488	0.366	0.355	0.354	0.354
`audiobeam`	1.000	1.000	1.000	0.487	0.363	0.350
`cjpeg_transupp`	1.000	1.000	1.000	1.000	1.000	0.413
`cjpeg_wrbmp`	1.000	1.000	0.613	0.337	0.309	0.307
`dijkstra`	1.000	1.000	1.000	1.000	1.000	1.000
`epic`	1.000	1.000	1.000	1.000	1.000	1.000
`g723_enc`	1.000	1.000	1.000	0.537	0.376	0.348
`gsm_dec`	1.000	1.000	1.000	1.000	0.419	0.335
`gsm_encode`	1.000	1.000	1.000	0.390	0.347	0.342
`h264_dec`	1.000	1.000	1.000	0.366	0.281	0.271
`huff_dec`	1.000	1.000	1.000	0.535	0.382	0.356
`mpeg2`	1.000	1.000	1.000	1.000	1.000	1.000
`ndes`	1.000	1.000	0.544	0.381	0.351	0.349
`petrinet`	1.000	0.474	0.377	0.367	0.366	0.366
`rijndael_dec`	1.000	1.000	1.000	1.000	1.000	1.000
`rijndael_enc`	1.000	1.000	1.000	1.000	0.614	0.394
`statemate`	1.000	1.000	1.000	0.412	0.372	0.368
`susan`	1.000	1.000	1.000	1.000	1.000	0.611
—**average**—	0.951	0.802	0.659	0.509	0.435	0.393
	0.594					

Table 10.5.: Arrival curve values calculated at the granularity of program runs for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$ assuming a relative minimum inter-start time of 0.5. The presented values are normalized to the respective interval lengths.

In the following sections, we demonstrate that an incorporation of the minimum inter-start time during implicit subpath enumeration (cf. Section 10.2) further decreases the average of the normalized arrival curve values (i.e. below 0.594, cf. Table 10.5).

10.3.2. Incorporation during Implicit Subpath Enumeration

In this section, we present a system property arguing about the minimum inter-start time $\mathit{MIST}_{prog_{C_j}}$ of a program $prog_{C_j}$. The presented system property is lifted to the level of approximation of implicit subpath enumeration in order to prune infeasible abstract traces during the calculation of arrival curve values for core C_j. Recall that, for simplicity, we assume an experimental setup in which every core C_j repeatedly executes a single program $prog_{C_j}$.

The system property that we exploit argues about the interval of positions of a concrete trace starting from including the first occurrence of event $\mathit{Start}_{prog_{C_j},C_j}$ up to excluding the last occurrence of event $\mathit{Start}_{prog_{C_j},C_j}$. This interval is referred to as *maximum between-starts interval* of concrete trace t ($\mathit{MBSI}(t)$). Note that, in case concrete trace t has less than two occurrence of event $\mathit{Start}_{prog_{C_j},C_j}$, the interval $\mathit{MBSI}(t)$ is empty.

$$\begin{aligned}\mathit{MBSI}(t) = \{x \in \mathbb{N}_{<len(t)} \;|[\exists lte \in \mathbb{N}_{\leq x} : \mathit{Start}_{prog_{C_j},C_j}(t, lte)] \wedge \\ [\exists gt \in \mathbb{N}_{>x} \cap \mathbb{N}_{<len(t)} : \mathit{Start}_{prog_{C_j},C_j}(t, gt)]\}\end{aligned} \tag{10.58}$$

The minimum inter-start time $\mathit{MIST}_{prog_{C_j}}$ (cf. equation (10.56)) guarantees a minimal distance between subsequent occurrences of event $\mathit{Start}_{prog_{C_j},C_j}$. This is exploited by system property $P_{\mathit{MIST}_{prog_{C_j}},C_j}$ in order to provide a lower bound on the number of cycle transition at the positions of concrete trace t that are spanned by interval $\mathit{MBSI}(t)$.

$$P_{\mathit{MIST}_{prog_{C_j}},C_j}(t) \Leftrightarrow \sum_{x \in \mathit{MBSI}(t)} \mathit{Start}_{prog_{C_j},C_j}(t, x) \cdot \mathit{MIST}_{prog_{C_j}} \leq \sum_{x \in \mathit{MBSI}(t)} \mathit{Cycle}(t, x) \tag{10.59}$$

The system property $P_{\mathit{MIST}_{prog_{C_j}},C_j}$ is lifted to the level of approximation of subpaths through a graph as follows.

$$\begin{aligned}\widehat{\mathit{MBSI}^{path}}(\widehat{p}) = \{x \in \mathbb{N}_{<len(\widehat{p})} \;|[\exists lte \in \mathbb{N}_{\leq x} : \widehat{w\mathit{Start}^{LB}_{prog_{C_j},C_j}}(\widehat{p}, lte) > 0] \wedge \\ [\exists gt \in \mathbb{N}_{>x} \cap \mathbb{N}_{<len(\widehat{p})} : \widehat{w\mathit{Start}^{LB}_{prog_{C_j},C_j}}(\widehat{p}, gt) > 0]\}\end{aligned} \tag{10.60}$$

$$\begin{aligned}\widehat{P^{path}_{\mathit{MIST}_{prog_{C_j}},C_j}}(\widehat{p}) \Leftrightarrow \sum_{x \in \widehat{\mathit{MBSI}^{path}}(\widehat{p})} \widehat{w\mathit{Start}^{LB}_{prog_{C_j},C_j}}(\widehat{p}, x) \cdot \mathit{MIST}_{prog_{C_j}} \\ \leq \sum_{x \in \widehat{\mathit{MBSI}^{path}}(\widehat{p})} \widehat{w\mathit{Cycle}}^{UB}(\widehat{p}, x)\end{aligned} \tag{10.61}$$

Due to time and space constraints, we omit a formal proof that the lifted property $\widehat{P^{path}_{\mathit{MIST}_{prog_{C_j}},C_j}}$ fulfills the soundness criteria for lifting system properties to the levels of approximation of sequences of abstract states and subpaths through a graph. Note, however, that such a proof would have to take into account that we only perform the calculation of arrival curve values on graphs at the granularity of basic blocks (cf. Section 6.4.4). In particular, we do not apply any graph transformations that merge chains of edges spanning across multiple basic block executions (cf. Section 6.4.6). If this was the case, the lifted property $\widehat{P^{path}_{\mathit{MIST}_{prog_{C_j}},C_j}}$ would have to be chosen in a more pessimistic way: in case $\widehat{\mathit{MBSI}^{path}}(\widehat{p}) \neq \emptyset$, the left-hand side of the inequation would

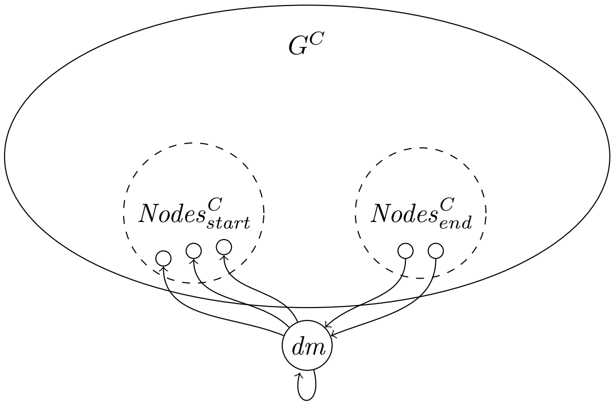


Figure 10.12.: For systems on which the program scheduler enforces minimum inter-start times, there can be filled-in clock cycles between subsequent program execution runs. Graph G^D (cf. Figure 10.8), from which the graphs used during implicit subpath enumeration are derived, reflects this by adding a self-edge to the dummy node *dm*. The self-edge corresponds to a single clock cycle and, thus, is referred to as *idle cycle*. The edge weights of the idle cycle assume that exactly one clock cycle is spent and no occurrence of event E happens.

have to consider the sum of all but the smallest element of the interval $\widehat{MBSI^{path}}(\widehat{p})$ and the right-hand side of the inequation would have to consider the sum of all elements of the interval $\widehat{MBSI^{path}}(\widehat{p}) \cup \{\max(\widehat{MBSI^{path}}(\widehat{p})) + 1\}$. A more detailed discussion, however, exceeds the scope of this thesis.

For systems on which the program scheduler does not enforce minimum inter-start times, the implicit subpath enumeration is performed on graphs derived from a graph G^D which assumes that a new execution run of a program starts as soon as an execution run of a (potentially different) program ends (cf. Figure 10.8 and Figure 10.10). For systems on which the program scheduler enforces minimum inter-start times, however, this is not sound as potentially filled-in clock cycles between two subsequent execution runs are not covered. To overcome this unsoundness, we add a self-edge to the dummy node *dm* of graph G^D. This is demonstrated in Figure 10.12. The self-edge corresponds to a single clock cycle between two subsequent execution runs and, thus, is referred to as *idle cycle*. The edge weights of the idle cycle reflect this by assuming that exactly one clock cycle is spent and no occurrence of event E happens.

$$idleCycle = (dm, dm) \tag{10.62}$$

$$\widehat{wCycle^{LB}}(idleCycle) = \widehat{wCycle^{UB}}(idleCycle) = 1 \tag{10.63}$$

$$\widehat{wE^{LB}}(idleCycle) = \widehat{wE^{UB}}(idleCycle) = 0 \tag{10.64}$$

The property $\widehat{P^{path}_{MIST_{prog_{C_j}},C_j}}$ on subpaths of the graph (cf. equation (10.61)) is lifted to the level of approximation of the implicit enumeration of subpaths as follows.

$$\begin{aligned}\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j}}((tt, is, ie)) \Leftrightarrow & [-1 + \sum_{edg \in Edges^F} tt(edg) \cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)] \cdot MIST_{prog_{C_j}} \\ & \leq \sum_{edg \in Edges^F} tt(edg) \cdot \widehat{wCycle^{UB}}(edg)\end{aligned} \tag{10.65}$$

Note that the lifted property $\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j}}$ as presented in equation (10.65) is overly pessimistic compared to the property $\widehat{P^{path}_{MIST_{prog_{C_j}},C_j}}$. This pessimism is due to the implicit (sub)path enumeration inherently approximating away the order of the edges in the described (sub)paths (cf. Section 5.4). The left-hand side of the inequation of property $\widehat{P^{path}_{MIST_{prog_{C_j}},C_j}}$ is still modeled precisely by the left-hand side of the inequation of property $\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j}}$: the number of program start events in the interval $\widehat{MBSI^{path}}(\widehat{p})$ of any described subpath $\widehat{p}$ is safely and precisely bounded by subtracting one from the lower bound on the number of occurrences of program start events along the implicit path. The right-hand side of the inequation of property $\widehat{P^{path}_{MIST_{prog_{C_j}},C_j}}$, however, is only imprecisely modeled by the right-hand side of the inequation of property $\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j}}$: as the order of the edges is approximated away in the implicit subpath, it is not clear how often an edge of a described subpath $\widehat{p}$ is taken within the interval $\widehat{MBSI^{path}}(\widehat{p})$ and, thus, every occurrence of an edge in the implicit path is pessimistically assumed to describe a subpath position in the interval $\widehat{MBSI^{path}}(\widehat{p})$. In Section 10.3.3, we demonstrate how to overcome this pessimism by simultaneously performing two implicit (sub)path enumerations.

Further note that the lifted property $\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j}}$ as presented in equation (10.65) is trivially lifted to the level of approximation of implicit subpath enumeration without binary variables (cf. Section 10.2.4) as it does not rely on which edge starts respectively ends the implicit path (cf. Section 9.4).

Finally, note that the lifted property $\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j}}$ argues about an upper bound on the number of clock cycles (i.e. $\widehat{wCycle}^{UB}$). The constraint encoding the interval length (cf. equation (10.29) respectively equation (10.44)), in contrast, argues about a lower bound on the number of clock cycles (i.e. $\widehat{wCycle}^{LB}$). In order to not let this consideration of different bound directions in different constraints lead to a reduced precision, one would typically resort to a full edge-weight-sensitivity of the lower bound on the number of clock cycles during graph construction (cf. Section 6.4.4) so that, for each edge of the resulting graph, the lower bound on the number of clock cycles and the upper bound on the number of clock cycles coincides. However, we expect that this additional increase of the edge-weight-sensitivity would lead to a blow-up of the resulting graph that would render the implicit subpath enumeration unmanageable with respect to its computational complexity. In order to not suffer from the graph blow-up and to still enjoy the precision of a full edge-weight-sensitivity of the lower bound on the number of clock cycles, we perform the implicit subpath enumeration on a graph that is edge-weight-insensitive with respect to the lower bound on the number of clock cycles and we replace the occurrences of $\widehat{wCycle}^{UB}$ in equation (10.65) by $\widehat{wCycle}^{LB}$. Intuitively, this is sound because the idle cycle (cf. Figure 10.12) enumerates all possible numbers of clock cycles and, thus, covers the same cases as a full edge-weight-sensitivity of the lower bound on the number of clock cycles. A more detailed discussion of this implementation trick and a formal soundness proof, however, exceed the scope of this thesis.

We experimentally evaluate the presented calculation method in combination with an implicit subpath enumeration with binary variables ($Bound^{impli}_{1,E}(l)$, cf. Section 10.2.3) for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The implicit enumeration of subpaths is performed on graphs that are fully node-sensitive at basic block boundaries, node-insensitive inside of basic blocks, and fully edge-weight-sensitive with respect to the upper bound on the number of occurrences of event E (cf. Section 6.4.4). As in the previous experiments with implicit subpath enumeration, we enforce a time limit of ten minutes per calculation of an arrival curve value. In case the ILP solver performing the implicit enumeration is not finished after this time limit, we use the best upper bound that the solver has calculated so far.

Table 10.6 lists the resulting arrival curve values normalized to the respective interval lengths. On average (geometric mean), the calculated curve values are 0.582 times as high as the corresponding interval lengths, which is only slightly better than for the curve value calculation at program granularity incorporating the minimum inter-start time (0.594, cf. Table 10.5). This corresponds to an average improvement of two percent compared to the curve value calculation at program granularity ($0.582/0.594 = 0.98$).

Note, however, that the average improvement of precision of an implicit subpath enumeration compared to a calculation at the granularity of program runs is more significant for smaller interval lengths. For an interval length of 10^4, e.g., the average factor is 0.868 compared to 0.951 for a calculation at the granularity of program runs. This corresponds to an average improvement of 8.7 percent compared to the curve value calculation at program granularity ($0.868/0.951 = 0.913$). This advantage of implicit subpath enumeration for small interval lengths can be explained by taking a closer look at the main source of imprecision that arises for the calculation at program granularity in the context of the current experiment. One of the major sources of imprecision of a calculation at program granularity is that the maximal amount of interference an execution run of a given program might create can typically not coincide with the execution run only taking the BCET of the program (cf. discussion at the end of Section 10.1). In combination with a relative minimum inter-start time of 0.5, this source of imprecision does no longer exist as each execution run is forced to effectively at least take an amount of time that is larger than the WCET bound assuming the absence of interference (cf. equation (10.57)). Thus, the major remaining source of imprecision is the pathological worst-case distribution of event occurrences that has to be considered during a calculation at program granularity (cf. Figure 10.3 on page 174). The imprecision stemming from this worst-case distribution, however, is typically relatively small compared to the interval length for interval lengths that are significantly larger than the minimum inter-start time of the program under consideration (cf. Figure 10.7 on page 179). This means that, for interval lengths that are significantly larger than the minimum inter-start time of the program under consideration, the relative gain in precision achieved by performing an implicit subpath enumeration is negligible.

The experiment takes 14 hours, 58 minutes, and 37.16 seconds. On average, the calculation of arrival curve values for a benchmark takes 10.82 times as long as the corresponding calculations at the granularity of program runs without minimum inter-start time (cf. Figure B.30 on page 347 of Appendix B). The memory consumption compared to the corresponding calculations at the granularity of program runs without minimum inter-start time increases by an average factor of 4.86 (cf. Figure B.31 on page 348 of Appendix B).

Note that the computational complexity and the memory consumption of implicit subpath enumeration are significantly increased by additionally incorporating a minimum inter-start time (cf. experimental results reported in Section 10.2.3). In Section 10.4, we sketch a more modular calculation method for values on arrival curves. We expect that the incorporation of minimum inter-start times in this modular calculation method does not significantly increase the computational complexity or the memory consumption.

In the following section, we explore a more precise approach to incorporate minimum inter-start times during implicit subpath enumeration.

10.3.3. More Precise Incorporation during Implicit Subpath Enumeration

As mentioned in Section 10.3.2, the level of approximation of implicit subpath enumeration does not directly permit a precise lifting of the property $\widehat{P^{path}_{MIST_{prog_{C_j}},C_j}}$. Intuitively, the problem is that the implicit subpath enumeration approximates away the order of edges in a described subpath $\widehat{p}$

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	0.927	0.923	0.453	0.369	0.362	0.361
`digital_stopwatch`	0.951	0.942	0.772	0.401	0.343	0.338
`es_lift`	0.959	0.957	0.482	0.372	0.365	0.364
`flight_control`	0.955	0.915	0.909	0.457	0.364	0.357
`pilot`	0.858	0.848	0.580	0.377	0.352	0.350
`roboDog`	0.952	0.950	0.663	0.395	0.374	0.372
`trolleybus`	0.966	0.962	0.921	0.422	0.364	0.360
`lift`	0.827	0.812	0.810	0.806	0.403	0.350
`powerwindow`	0.941	0.937	0.937	0.933	0.604	0.377
`binarysearch`	0.421	0.366	0.362	0.361	0.361	0.361
`bsort`	0.842	0.837	0.836	0.429	0.364	0.352
`complex_updates`	0.832	0.388	0.363	0.360	0.360	0.359
`countnegative`	0.832	0.648	0.389	0.371	0.368	0.368
`fft`	0.899	0.896	0.859	0.812	0.774	0.758
`filterbank`	0.840	0.786	0.778	0.727	0.538	0.349
`fir2dim`	0.837	0.511	0.370	0.349	0.348	0.347
`iir`	0.466	0.379	0.369	0.368	0.368	0.368
`insertsort`	0.857	0.510	0.367	0.356	0.355	0.355
`jfdctint`	0.829	0.442	0.348	0.343	0.342	0.342
`lms`	0.871	0.867	0.774	0.422	0.347	0.338
`ludcmp`	0.874	0.603	0.382	0.352	0.348	0.348
`matrix1`	0.796	0.791	0.416	0.353	0.348	0.347
`md5`	0.898	0.869	0.866	0.865	0.860	0.529
`minver`	0.880	0.512	0.371	0.348	0.346	0.346
`pm`	0.938	0.917	0.884	0.838	0.752	1.000
`prime`	0.823	0.441	0.264	0.250	0.247	0.247
`sha`	0.901	0.851	0.845	0.828	0.492	0.367
`st`	0.669	0.624	0.511	0.323	0.306	0.304
`adpcm_dec`	0.858	0.539	0.368	0.356	0.355	0.354
`adpcm_enc`	0.851	0.545	0.372	0.356	0.354	0.354
`audiobeam`	0.934	0.931	0.930	0.599	0.376	0.351
`cjpeg_transupp`	0.911	0.849	0.842	0.842	0.831	0.476
`cjpeg_wrbmp`	0.771	0.734	0.705	0.368	0.312	0.307
`dijkstra`	0.907	0.902	0.902	0.902	0.894	0.892
`epic`	1.000	1.000	0.947	1.000	1.000	1.000
`g723_enc`	0.940	0.937	0.850	0.779	0.376	0.349
`gsm_dec`	0.918	0.803	0.783	0.714	0.524	0.335
`gsm_encode`	0.858	0.796	0.782	0.449	0.354	0.342
`h264_dec`	0.927	0.908	0.900	0.887	0.365	0.279
`huff_dec`	0.900	0.892	0.879	0.764	0.382	0.359
`mpeg2`	1.000	1.000	1.000	1.000	1.000	1.000
`ndes`	0.961	0.877	0.817	0.381	0.354	0.349
`petrinet`	0.933	0.474	0.379	0.368	0.367	0.366
`rijndael_dec`	0.988	0.981	0.980	0.980	0.980	0.980
`rijndael_enc`	0.988	0.979	0.978	0.978	0.614	0.409
`statemate`	0.947	0.941	0.941	0.481	0.378	0.368
`susan`	1.000	1.000	1.000	1.000	1.000	1.000
—**average**—	0.868	0.748	0.637	0.520	0.443	0.410
			0.582			

Table 10.6.: Arrival curve values calculated by implicit subpath enumeration for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$ assuming a relative minimum inter-start time of 0.5. The presented values are normalized to the respective interval lengths.

and, thus, it is not clear how often an edge of subpath $\widehat{p}$ is taken within the interval $\widehat{MBSI^{path}}(\widehat{p})$. In this subsection, we demonstrate how to overcome this inherent lack of expressiveness by simultaneously performing two implicit (sub)path enumerations.

To this end, we start by taking a closer look at the part of a concrete trace t about which the concrete system property $P_{MIST_{prog_{C_j}},C_j}$ argues. The helper function *MBSIcut* returns a set containing the cut-out sub sequence of concrete trace t featuring all positions of the interval $\mathit{MBSI}(t)$. In case the interval $\mathit{MBSI}(t)$ is empty, the helper function returns a set of sequences that only contain a single initial state of the program.

$$\begin{aligned} \mathit{MBSIcut}(t) = &\{t' \in \mathit{Sequences} \mid \mathit{MBSI}(t) \neq \emptyset \wedge \mathit{len}(t') = |\mathit{MBSI}(t)| \wedge \\ &\quad \forall x \in \mathbb{N}_{\leq len(t')} : t'(x) = t(\min(\mathit{MBSI}(t)) + x)\} \cup \\ &\{t' \in \mathit{Sequences} \mid \mathit{MBSI}(t) = \emptyset \wedge \mathit{len}(t') = |\mathit{MBSI}(t)| \wedge \\ &\quad t'(0) \in \mathit{InitStates}_{prog_{C_j},C_j}\} \end{aligned} \tag{10.66}$$

We use the helper function *MBSIcut* to define the set containing the cut-out sub sequences of all concrete traces. Intuitively, each member of this set corresponds to a sequence of zero or more complete execution runs of program $prog_{C_j}$ on core C_j including potentially filled-in cycles after each complete run due to the enforcing of the minimum inter-start time.

$$\mathit{ExecRunsMult}_{prog_{C_j},C_j} = \bigcup_{t \in ExecRuns_{prog_{C_i},C_i}} \mathit{MBSIcut}(t) \tag{10.67}$$

The set $\mathit{ExecRunsMult}_{prog_{C_j},C_j}$ is safely overapproximated by the set $\widehat{\mathit{ImplicitMult}}^D$ considered during a slight variant of an implicit path enumeration (cf. equation (5.81) in Section 5.4) based on graph G^D (cf. Figure 10.12). Note that the main difference to the original implicit path enumeration (set $\widehat{\mathit{Implicit}}^D$, cf. equation (5.81)) is that, this time, we also consider multiple consecutive complete program execution runs instead of only up to one.

$$\begin{aligned} \widehat{\mathit{ImplicitMult}}^D = &\{(\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd}) \in (\mathit{Edges} \to \mathbb{N}) \times (\mathit{Edges} \to \{0,1\})^2 \mid \\ &\quad [\ \forall e \in \mathit{Edges}^D : \mathit{isStart}(e) \leq \mathit{timesTaken}(e)\ ] \wedge \\ &\quad [\ \forall e \in \mathit{Edges}^D : \mathit{isEnd}(e) \leq \mathit{timesTaken}(e)\ ] \wedge \\ &\quad \sum_{e \in Edges^D} \mathit{isStart}(e) = \sum_{e \in Edges} \mathit{isEnd}(e) \wedge \\ &\quad \sum_{e \in (Edges^D \setminus Edges^D_{start})} \mathit{isStart}(e) = 0 \wedge \\ &\quad \sum_{e \in (Edges^D \setminus Edges^D_{end})} \mathit{isEnd}(e) = 0 \wedge \\ &\quad [\ \mathit{Nodes}^D_{start} \cap \mathit{Nodes}^D_{end} = \emptyset \Rightarrow \sum_{e \in Edges^D} \mathit{isStart}(e) \geq 1\ ] \wedge \\ &\quad [\ \forall \mathit{node} \in \mathit{Nodes}^D : \\ &\qquad \sum_{e_{in} \in inEdges(node)} [\mathit{timesTaken}(e_{in}) - \mathit{isEnd}(e_{in})] \\ &\qquad = \sum_{e_{out} \in outEdges(node)} [\mathit{timesTaken}(e_{out}) - \mathit{isStart}(e_{out})]\ ] \\ &\} \end{aligned} \tag{10.68}$$

There shall be a set $\mathit{PropMult}_{prog_{C_j},C_j}$ of system properties that hold for every member of set $\mathit{ExecRunsMult}_{prog_{C_j},C_j}$.

$$\forall t \in \mathit{ExecRunsMult}_{prog_{C_j},C_j} : \forall P_k \in \mathit{PropMult}_{prog_{C_j},C_j} : P_k(t) \tag{10.69}$$

As an example, consider the system property $P_{MIST_{prog_{C_j}},C_j,Mult} \in \mathit{PropMult}_{prog_{C_j},C_j}$. It is a direct consequence of the system property $P_{MIST_{prog_{C_j}},C_j}$ on concrete traces and the definition of the set $\mathit{ExecRunsMult}_{prog_{C_j},C_j}$.

$$P_{MIST_{prog_{C_j}},C_j,Mult}(t) \Leftrightarrow \sum_{x\in\mathbb{N}_{<len(t)}} \mathit{Start}_{prog_{C_j},C_j}(t,x) \cdot \mathit{MIST}_{prog_{C_j}} \leq \sum_{x\in\mathbb{N}_{<len(t)}} \mathit{Cycle}(t,x) \tag{10.70}$$

The system property $P_{MIST_{prog_{C_j}},C_j,Mult}$ is lifted to the level of approximation of implicit path enumeration as follows.

$$\begin{aligned}\widehat{P^{impli}_{MIST_{prog_{C_j}},C_j,Mult}}((tt,is,ie)) \Leftrightarrow & \sum_{edg\in Edges^D} tt(edg) \cdot w\widehat{Start}^{LB}_{prog_{C_j},C_j}(edg) \cdot \mathit{MIST}_{prog_{C_j}} \\ & \leq \sum_{edg\in Edges^D} tt(edg) \cdot w\widehat{Cycle}^{UB}(edg)\end{aligned} \tag{10.71}$$

In a similar way as described in Section 10.3.2, in our implementation, we replace the occurrences of $w\widehat{Cycle}^{UB}$ in equation (10.71) by $w\widehat{Cycle}^{LB}$ in order to achieve a reasonable precision on a graph G^D that is edge-weight-insensitive with respect to the lower bound on the number of clock cycles.

Note that some classical control flow properties that are typically exploited during the calculation of per-execution-run event bounds—as e.g. the classical relative loop bound properties [Li and Malik, 1995] (cf. equation (10.16))—are also contained in $\mathit{PropMult}_{prog_{C_j},C_j}$.

Lifted versions of the properties in $\mathit{PropMult}_{prog_{C_j},C_j}$ are used for pruning infeasible members of $\widehat{\mathit{ImplicitMult}}^D$.

$$\mathit{Less}\widehat{\mathit{ImplicitMult}}^D = \{(tt,is,ie) \in \widehat{\mathit{ImplicitMult}}^D \mid \forall P_k \in \mathit{PropMult}_{prog_{C_j},C_j} : \widehat{P^{impli}_k}((tt,is,ie))\} \tag{10.72}$$

The actual calculation of values on arrival curves is performed on the cross product of $\mathit{Less}\widehat{\mathit{Implicit}}^F_{\leq l}$ (cf. Section 10.2.3) and $\mathit{Less}\widehat{\mathit{ImplicitMult}}^D$. This means that the enumeration simultaneously considers two implicit (sub)paths.

$$((tt,is,ie),(tt',is',ie')) \in \mathit{Less}\widehat{\mathit{Implicit}}^F_{\leq l} \times \mathit{Less}\widehat{\mathit{ImplicitMult}}^D \tag{10.73}$$

As described in Section 10.2.3, the objective functions for the curve value calculations only argue about the component (tt,is,ie) from set $\mathit{Less}\widehat{\mathit{Implicit}}^F_{\leq l}$. In order to anyway potentially result in more precise curve values due to the simultaneous enumeration and the more precisely lifted MIST property on component (tt',is',ie'), we add constraints relating both components. First, we require that the implicit subpath (tt',is',ie') is completely contained in the implicit subpath (tt,is,ie). Thus, no edge of graph G^D shall be taken by implicit subpath (tt',is',ie') more often than by implicit subpath (tt,is,ie).

$$\forall edg \in Edges^D : tt'(edg) \leq tt(edg) \tag{10.74}$$

Additionally, we require the sum of the guaranteed occurrences of the program start event in implicit subpath (tt', is', ie') to be one smaller than the sum of the guaranteed occurrences of the program start event in implicit subpath (tt, is, ie). The only exception is that the sum of the guaranteed occurrences of the program start event in implicit subpath (tt', is', ie') shall be zero in case the sum of the guaranteed occurrences of the program start event in implicit subpath (tt, is, ie) is zero. Note that this constraint is inspired by a corresponding relation between the numbers of occurrences of the program start event in *MBSIcut*(t) and t at the level of concrete traces.

$$\begin{aligned}&\sum_{edg\in Edges^D} tt'(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)\\&\quad=\max\left(-1+\sum_{edg\in Edges^D} tt(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg),0\right)\end{aligned}\tag{10.75}$$

The maximum operation in constraint (10.75) is typically not directly supported by ILP solvers. Thus, we reformulate the constraint in the following way by making use of a binary helper variable *containsProgStart*. The helper variable shall be one if and only if the sum of the guaranteed occurrences of the program start event in implicit subpath tt is greater than zero.

$$\begin{aligned}&\sum_{edg\in Edges^D} tt'(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)\\&\quad=-containsProgStart+\sum_{edg\in Edges^D} tt(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)\end{aligned}\tag{10.76}$$

$$containsProgStart \Leftrightarrow \sum_{edg\in Edges^D} tt(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)>0\tag{10.77}$$

The equivalence requirement of equation (10.77) is also typically not directly supported by ILP solvers. Thus, we simulate the equivalence requirement by the following two inequations. The interval length l is used as a Big M in the second inequation.

$$containsProgStart \leq \sum_{edg\in Edges^D} tt(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)\tag{10.78}$$

$$containsProgStart\cdot l \geq \sum_{edg\in Edges^D} tt(edg)\cdot \widehat{wStart^{LB}_{prog_{C_j},C_j}}(edg)\tag{10.79}$$

Note that we only formally derived the simultaneous consideration of multiple implicit paths for implicit path enumeration with binary variables (cf. Section 5.4). A corresponding derivation for implicit path enumeration without binary variables (cf. Section 9.4) can be performed analogously.

Further note that the general idea of simultaneously considering multiple implicit paths and relating them is not new. ILP-based approaches to cache modeling [Li et al., 1995, 1996] simultaneously consider an implicit path through the control flow graph and an implicit path through the graph of concrete cache states of the program under analysis. Due to the huge size of the graph of concrete cache states for real-world programs executed on realistic hardware platforms, however, these approaches suffer from a high computational complexity [Lv et al., 2016].

We experimentally evaluate the presented calculation method in combination with an implicit subpath enumeration with binary variables ($Bound^{impli}_{1,E}(l)$, cf. Section 10.2.3) for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$. The implicit enumeration of subpaths is performed on graphs that are fully node-sensitive at basic block boundaries, node-insensitive inside of basic blocks, and fully edge-weight-sensitive with respect to the upper bound on the number of occurrences of event E (cf. Section 6.4.4). As in the previous experiments with implicit subpath enumeration, we enforce a time limit of ten minutes per calculation of an arrival curve value. In case the ILP solver performing the implicit enumeration is not finished after this time limit, we use the best upper bound that the solver has calculated so far.

Table 10.7 lists the resulting arrival curve values normalized to the respective interval lengths. On average (geometric mean), the calculated curve values are 0.569 times as high as the corresponding interval lengths, which is slightly better than the implicit subpath enumeration incorporating a minimum inter-start time as presented in Section 10.3.2 (0.582, cf. Table 10.6). This corresponds to an average improvement of 4.2 percent compared to the curve value calculation at program granularity ($0.569/0.594 = 0.958$, cf. Table 10.5). The simple incorporation of the minimum inter-start time presented in Section 10.3.2 only results in an average improvement of two percent. Thus, the detailed modeling of the minimum inter-start time by a simultaneous consideration of two implicit paths can effectively increase the precision of the calculated arrival curve values.

The experiment takes 16 hours, 32 minutes, and 5.37 seconds. On average, the calculation of arrival curve values for a benchmark takes 16.21 times as long as the corresponding calculations at the granularity of program runs without minimum inter-start time (cf. Figure B.32 on page 349 of Appendix B). The memory consumption compared to the corresponding calculations at the granularity of program runs without minimum inter-start time increases by an average factor of 5.53 (cf. Figure B.33 on page 350 of Appendix B). These results demonstrate that the more precise modeling of the minimum inter-start time during implicit subpath enumeration comes at the cost of an increased computational complexity and memory consumption (cf. Section 10.3.2 for the corresponding experiment results for the less precise modeling of the minimum inter-start time during implicit subpath enumeration).

Comparing the normalized curve values of Table 10.7 with the corresponding normalized curve values presented in Section 10.3.1 (Table 10.5 on page 198), we see that the curve values calculated by implicit subpath enumeration are in some cases significantly greater than the curve values calculated at the granularity of program runs—although they are on average smaller. This is e.g. the case for benchmark `susan` in combination with interval length 10^9. We think that these less precise results for implicit subpath enumeration are due to the high computational complexity of implicit subpath enumeration and the corresponding time limit of ten minutes that we enforce per calculation of an arrival curve value. Intuitively, after ten minutes, the implicit subpath enumeration has not yet determined an upper bound on the arrival curve value that is more precise than an arrival curve value calculated at the granularity of program runs. In order to further increase the precision of the calculated arrival curve values, in the following section, we explore combinations of implicit subpath enumeration and a calculation at the granularity of program runs.

10.3.4. Combining a Curve Value Calculation at the Granularity of Program Runs with Implicit Subpath Enumeration

As a starting point, we combine a curve value calculation at the granularity of program runs with implicit subpath enumeration by calculating curve values via both methods and subsequently taking the minimum of the resulting curve values. This means that, for an experimental evaluation, we perform the curve value calculations presented in Section 10.3.1 and Section 10.3.3. Note, however, that the micro-architectural analysis is only performed once per benchmark. Subsequently, we select the minimum of the two resulting curve values.

Table 10.8 lists the resulting arrival curve values normalized to the respective interval lengths. On average (geometric mean), the calculated curve values are 0.563 times as high as the corresponding interval lengths, which is slightly better than the implicit subpath enumeration presented in Section 10.3.3 (0.569, cf. Table 10.7). This corresponds to an average improvement of 5.2 percent compared to the curve value calculation at program granularity ($0.563/0.594 = 0.948$,

	10^4	10^5	10^6	10^7	10^8	10^9
cruise_control	0.927	0.923	0.451	0.369	0.362	0.361
digital_stopwatch	0.951	0.942	0.772	0.369	0.342	0.338
es_lift	0.959	0.957	0.413	0.372	0.364	0.364
flight_control	0.955	0.915	0.909	0.404	0.363	0.356
pilot	0.858	0.848	0.435	0.363	0.351	0.350
roboDog	0.952	0.950	0.498	0.386	0.373	0.372
trolleybus	0.966	0.962	0.921	0.411	0.364	0.359
lift	0.827	0.812	0.810	0.672	0.370	0.350
powerwindow	0.941	0.937	0.937	0.933	0.453	0.377
binarysearch	0.399	0.366	0.362	0.361	0.361	0.361
bsort	0.842	0.837	0.836	0.429	0.357	0.352
complex_updates	0.555	0.388	0.362	0.360	0.360	0.359
countnegative	0.832	0.486	0.383	0.369	0.368	0.368
fft	0.899	0.896	0.857	0.812	0.774	0.758
filterbank	0.840	0.786	0.778	0.727	0.403	0.336
fir2dim	0.837	0.471	0.358	0.349	0.347	0.347
iir	0.419	0.372	0.368	0.368	0.368	0.368
insertsort	0.857	0.423	0.366	0.356	0.355	0.355
jfdctint	0.829	0.387	0.348	0.342	0.342	0.342
lms	0.871	0.867	0.774	0.375	0.342	0.338
ludcmp	0.874	0.535	0.362	0.350	0.348	0.348
matrix1	0.796	0.791	0.392	0.352	0.348	0.347
md5	0.898	0.869	0.866	0.865	0.860	0.443
minver	0.880	0.476	0.359	0.347	0.346	0.346
pm	0.938	1.000	1.000	1.000	1.000	0.358
prime	0.823	0.441	0.264	0.248	0.247	0.247
sha	0.901	0.851	0.845	0.828	0.443	0.358
st	0.669	0.624	0.429	0.318	0.305	0.304
adpcm_dec	0.858	0.438	0.368	0.356	0.354	0.354
adpcm_enc	0.851	0.503	0.370	0.355	0.354	0.354
audiobeam	0.934	0.931	0.930	0.496	0.364	0.350
cjpeg_transupp	0.911	0.849	0.842	0.842	1.000	1.000
cjpeg_wrbmp	0.771	0.734	0.613	0.337	0.309	0.307
dijkstra	0.907	0.902	0.902	0.902	0.894	0.892
epic	1.000	1.000	1.000	1.000	1.000	1.000
g723_enc	0.940	0.937	0.850	0.538	0.374	0.347
gsm_dec	0.918	0.803	0.783	0.714	0.419	0.329
gsm_encode	0.858	0.796	0.782	0.386	0.348	0.342
h264_dec	0.927	0.909	0.900	0.869	0.334	0.276
huff_dec	0.900	0.892	0.879	0.509	0.382	0.356
mpeg2	1.000	1.000	1.000	1.000	1.000	1.000
ndes	0.961	0.877	0.545	0.379	0.351	0.349
petrinet	0.933	0.474	0.375	0.367	0.366	0.366
rijndael_dec	0.987	0.981	0.980	0.980	0.980	0.980
rijndael_enc	0.985	0.979	0.978	0.978	0.614	0.393
statemate	0.947	0.941	0.941	0.412	0.372	0.368
susan	1.000	1.000	1.000	1.000	1.000	1.000
—average—	0.857	0.731	0.617	0.500	0.436	0.404
	0.569					

Table 10.7.: Arrival curve values calculated by implicit subpath enumeration for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$ incorporating a relative minimum inter-start time of 0.5 in a more precise way. The presented values are normalized to the respective interval lengths.

cf. Table 10.5). Implicit subpath enumeration results in an average improvement of at most 4.2 percent (cf. Section 10.3.3). Thus, combinations of implicit subpath enumeration and a calculation at the granularity of program runs can further increase the precision of the calculated curve values.

The experiment takes 16 hours, 41 minutes, and 2.54 seconds (which is around as long as the experiment in Section 10.3.3). On average, the calculation of arrival curve values for a benchmark takes 16.31 times as long as the corresponding calculations at the granularity of program runs without minimum inter-start time (cf. Figure B.34 on page 351 of Appendix B). The memory consumption compared to the corresponding calculations at the granularity of program runs without minimum inter-start time increases by an average factor of 5.60 (cf. Figure B.35 on page 352 of Appendix B).

As a next step, we would like to find out whether the implicit subpath enumeration presented in Section 10.3.3 can be further improved by upper-bounding its objective with a curve value $\alpha_E^{\mathit{prog\text{-}gran}}(l)$ calculated as presented in Section 10.3.1. This means that, first, we perform a curve value calculation at the granularity of program runs. The resulting curve value is subsequently incorporated during the generation of the implicit subpath enumeration. To this end, we adapt the objective of implicit subpath enumeration by introducing a helper variable $\mathit{occurrences}_E$.

$$\mathit{Bound}_{1,E}^{\mathit{pg,impli}}(l) = \max_{((tt,is,ie),(tt',is',ie'),\mathit{occurrences}_E)\in \widehat{\mathit{LessImplicit}}_{\leq l}^{F}\times \widehat{\mathit{LessImplicit}}^{D}\times\mathbb{N}} \mathit{occurrences}_E \tag{10.80}$$

The helper variable is upper-bounded by the curve value $\alpha_E^{\mathit{prog\text{-}gran}}(l)$ and by the objective that is used during the calculation of the original curve value $\mathit{Bound}_{1,E}^{\mathit{impli}}(l)$ (cf. equation (10.34)).

$$\mathit{occurrences}_E \leq \alpha_E^{\mathit{prog\text{-}gran}}(l) \tag{10.81}$$

$$\mathit{occurrences}_E \leq \sum_{edg\in \mathit{Edges}^F} tt(edg)\cdot \widehat{wE^{UB}}(edg) \tag{10.82}$$

Table 10.9 lists the resulting arrival curve values normalized to the respective interval lengths. The normalized curve values are as precise as those listed in Table 10.8. Thus, in terms of precision, the incorporation of the additional upper bound in the implicit subpath enumeration does not pay off.

The experiment takes 17 hours, 33 minutes, and 0.16 seconds, which is almost one hour longer than the previous experiment. Thus, the overall computational complexity of the experiment is slightly increased compared to the previous experiment. On average, the calculation of arrival curve values for a benchmark takes 15.68 times as long as the corresponding calculations at the granularity of program runs without minimum inter-start time (cf. Figure B.36 on page 353 of Appendix B). The memory consumption compared to the corresponding calculations at the granularity of program runs without minimum inter-start time increases by an average factor of 5.50 (cf. Figure B.37 on page 354 of Appendix B). Note that these average increase factors are slightly reduced compared to the previous experiment. We follow that the computational complexity compared to the previous experiment is only increased for some of the benchmarks. These benchmarks, however, seem to dominate the overall runtime of the experiment.

This concludes the section presenting two possible approaches to combining a curve value calculation at the granularity of program runs with implicit subpath enumeration. In terms of precision, both approaches are equivalent. With respect to calculation runtime and memory consumption, it depends on the considered benchmark which of both approaches is more beneficial.

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	0.927	0.923	0.451	0.369	0.362	0.361
`digital_stopwatch`	0.951	0.942	0.772	0.369	0.342	0.338
`es_lift`	0.959	0.957	0.413	0.372	0.364	0.364
`flight_control`	0.955	0.915	0.909	0.404	0.363	0.356
`pilot`	0.858	0.848	0.435	0.363	0.351	0.350
`roboDog`	0.952	0.950	0.494	0.386	0.373	0.372
`trolleybus`	0.966	0.962	0.921	0.411	0.364	0.359
`lift`	0.827	0.812	0.810	0.672	0.370	0.350
`powerwindow`	0.941	0.937	0.937	0.933	0.453	0.377
`binarysearch`	0.399	0.366	0.362	0.361	0.361	0.361
`bsort`	0.842	0.837	0.836	0.429	0.357	0.352
`complex_updates`	0.554	0.388	0.362	0.360	0.360	0.359
`countnegative`	0.832	0.486	0.383	0.369	0.368	0.368
`fft`	0.899	0.896	0.857	0.812	0.774	0.758
`filterbank`	0.840	0.786	0.778	0.727	0.403	0.336
`fir2dim`	0.837	0.471	0.358	0.349	0.347	0.347
`iir`	0.419	0.372	0.368	0.368	0.368	0.368
`insertsort`	0.857	0.423	0.366	0.356	0.355	0.355
`jfdctint`	0.829	0.387	0.348	0.342	0.342	0.342
`lms`	0.871	0.867	0.774	0.375	0.342	0.338
`ludcmp`	0.874	0.535	0.362	0.350	0.348	0.348
`matrix1`	0.796	0.791	0.392	0.352	0.348	0.347
`md5`	0.898	0.869	0.866	0.865	0.860	0.440
`minver`	0.880	0.476	0.359	0.347	0.346	0.346
`pm`	0.938	1.000	1.000	1.000	0.534	0.348
`prime`	0.823	0.441	0.264	0.248	0.247	0.247
`sha`	0.901	0.851	0.845	0.828	0.443	0.358
`st`	0.669	0.624	0.429	0.318	0.305	0.304
`adpcm_dec`	0.858	0.438	0.365	0.355	0.354	0.354
`adpcm_enc`	0.851	0.488	0.366	0.355	0.354	0.354
`audiobeam`	0.934	0.931	0.930	0.487	0.363	0.350
`cjpeg_transupp`	0.911	0.849	0.842	1.000	1.000	0.413
`cjpeg_wrbmp`	0.771	0.734	0.613	0.337	0.309	0.307
`dijkstra`	0.907	0.902	0.902	0.902	0.894	0.892
`epic`	1.000	1.000	1.000	1.000	1.000	1.000
`g723_enc`	0.940	0.937	0.850	0.537	0.374	0.347
`gsm_dec`	0.918	0.803	0.783	0.714	0.419	0.329
`gsm_encode`	0.858	0.796	0.782	0.386	0.347	0.342
`h264_dec`	0.927	0.909	0.900	0.366	0.281	0.271
`huff_dec`	0.900	0.892	0.879	0.509	0.382	0.356
`mpeg2`	1.000	1.000	1.000	1.000	1.000	1.000
`ndes`	0.961	0.877	0.544	0.379	0.351	0.349
`petrinet`	0.933	0.474	0.375	0.367	0.366	0.366
`rijndael_dec`	0.987	0.981	0.980	0.980	0.980	0.980
`rijndael_enc`	0.985	0.979	0.978	0.978	0.614	0.393
`statemate`	0.947	0.941	0.941	0.412	0.372	0.368
`susan`	1.000	1.000	1.000	1.000	1.000	0.611
—**average**—	0.857	0.730	0.616	0.493	0.428	0.392
			0.563			

Table 10.8.: Arrival curve values calculated for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$ by taking the minimum of the curve values calculated by the approaches presented in Section 10.3.1 and Section 10.3.3. The presented values are normalized to the respective interval lengths.

	10^4	10^5	10^6	10^7	10^8	10^9
`cruise_control`	0.927	0.923	0.451	0.369	0.362	0.361
`digital_stopwatch`	0.951	0.942	0.772	0.369	0.342	0.338
`es_lift`	0.959	0.957	0.413	0.372	0.364	0.364
`flight_control`	0.955	0.915	0.909	0.404	0.363	0.356
`pilot`	0.858	0.848	0.435	0.363	0.351	0.350
`roboDog`	0.952	0.950	0.494	0.386	0.373	0.372
`trolleybus`	0.966	0.962	0.921	0.411	0.364	0.359
`lift`	0.827	0.812	0.810	0.672	0.370	0.350
`powerwindow`	0.941	0.937	0.937	0.933	0.464	0.377
`binarysearch`	0.399	0.366	0.362	0.361	0.361	0.361
`bsort`	0.842	0.837	0.836	0.429	0.357	0.352
`complex_updates`	0.554	0.388	0.362	0.360	0.360	0.359
`countnegative`	0.832	0.486	0.383	0.369	0.368	0.368
`fft`	0.899	0.896	0.857	0.812	0.774	0.758
`filterbank`	0.840	0.786	0.778	0.727	0.403	0.336
`fir2dim`	0.838	0.471	0.358	0.349	0.347	0.347
`iir`	0.419	0.372	0.368	0.368	0.368	0.368
`insertsort`	0.857	0.423	0.366	0.356	0.355	0.355
`jfdctint`	0.829	0.387	0.348	0.342	0.342	0.342
`lms`	0.871	0.867	0.774	0.375	0.342	0.338
`ludcmp`	0.874	0.535	0.362	0.350	0.348	0.348
`matrix1`	0.796	0.791	0.392	0.352	0.348	0.347
`md5`	0.897	0.869	0.866	0.865	0.860	0.440
`minver`	0.880	0.476	0.359	0.347	0.346	0.346
`pm`	1.000	1.000	1.000	1.000	0.534	0.348
`prime`	0.823	0.441	0.264	0.248	0.247	0.247
`sha`	0.901	0.851	0.845	0.828	0.443	0.358
`st`	0.669	0.624	0.429	0.318	0.305	0.304
`adpcm_dec`	0.858	0.438	0.365	0.355	0.355	0.354
`adpcm_enc`	0.851	0.488	0.366	0.355	0.354	0.354
`audiobeam`	0.934	0.931	0.930	0.487	0.363	0.350
`cjpeg_transupp`	0.911	0.849	0.842	0.842	1.000	0.413
`cjpeg_wrbmp`	0.771	0.734	0.613	0.337	0.309	0.307
`dijkstra`	0.907	0.902	0.902	0.902	0.894	0.892
`epic`	1.000	1.000	1.000	1.000	1.000	1.000
`g723_enc`	0.940	0.937	0.850	0.537	0.374	0.347
`gsm_dec`	0.918	0.803	0.783	0.714	0.419	0.329
`gsm_encode`	0.858	0.796	0.782	0.386	0.347	0.342
`h264_dec`	0.927	0.909	0.900	0.366	0.281	0.271
`huff_dec`	0.900	0.892	0.879	0.509	0.382	0.356
`mpeg2`	1.000	1.000	1.000	1.000	1.000	1.000
`ndes`	0.961	0.877	0.544	0.379	0.351	0.349
`petrinet`	0.933	0.474	0.375	0.367	0.366	0.366
`rijndael_dec`	0.990	0.981	0.980	0.980	0.980	0.980
`rijndael_enc`	0.986	0.979	0.978	0.978	0.614	0.393
`statemate`	0.947	0.941	0.941	0.412	0.372	0.368
`susan`	1.000	1.000	1.000	1.000	1.000	0.611
—**average**—	0.858	0.730	0.616	0.491	0.429	0.392
			0.563			

Table 10.9.: Arrival curve values calculated for the interval lengths in $\{10^4, 10^5, 10^6, 10^7, 10^8, 10^9\}$ by incorporating the curve value calculated by the approach presented in Section 10.3.1 as upper bound during the calculation presented in Section 10.3.3. The presented values are normalized to the respective interval lengths.

10.4. Sketch: A Program-Modular and Precise Calculation Method

We expect the computational complexity of a calculation of arrival curve values directly based on implicit subpath enumeration (cf. Section 10.2) to be unmanageable when dealing with multiple real-world programs executed on the same processor core (cf. Section 10.2.5). Moreover, even if the computational complexity is still bearable, the joint consideration of multiple programs on the same core during implicit subpath enumeration is inappropriate for a scenario with continuous development: In case only one of the programs of a processor core changes during development, the very costly calculations of arrival curve values (incorporating all programs of the core) have to be redone from scratch during the timing verification.

To overcome these drawbacks, this section sketches a more modular calculation of arrival curve values: The computationally complex aspects are calculated at a per-program level and the actual calculation of arrival curve values only uses the results of the per-program calculations. Yet, the sketched approach takes into account which part of a program can produce which amount of event occurrences and, thus, avoids the pessimism of a calculation at the granularity of program runs (cf. Section 10.1).

For the sake of readability, the sketch of the program-modular calculation method is structured into multiple subsections.

10.4.1. Paths through a Scheduling Graph

During a time interval of length l, the processor core that is considered during the calculation of arrival curve values can potentially execute different sequences of program execution runs. The possible sequences of program execution runs inherently depend on the non-preemptive scheduling strategy that is applied. In this subsection, we demonstrate how to safely overapproximate the possible sequences of program execution runs by the paths through a scheduling graph. Intuitively, a scheduling graph describes different sequences of program start events of the programs executed on a processor core C_j.

We start by introducing a *generic scheduling graph* that safely overapproximate the possible sequences of program execution runs in combination with any non-preemptive scheduling strategy. Each program $prog_{C_j}$ that is executed on the considered processor core C_j is represented in the generic scheduling graph by a directed edge $(nd_{sc,prog_{C_j}}, nd_{tg,prog_{C_j}})$ with a dedicated source node $nd_{sc,prog_{C_j}}$ and a dedicated target node $nd_{tg,prog_{C_j}}$. The edge representing a program $prog_{C_j}$ shall describe exactly one occurrence of program start event $Start_{prog_{C_j},C_j}$ and no occurrences of the start events of the other programs executed on core C_j.

$$\begin{aligned}
&\forall prog_{C_j} \in Programs_{C_j} : \\
&\quad \forall edg_{prog_{C_j}} \in \{(nd_{sc,prog_{C_j}}, nd_{tg,prog_{C_j}})\} : \\
&\qquad \widehat{wStart}^{UB}_{prog_{C_j},C_j}(edg_{prog_{C_j}}) = \widehat{wStart}^{LB}_{prog_{C_j},C_j}(edg_{prog_{C_j}}) = 1 \\
&\qquad \forall prog'_{C_j} \in Programs_{C_j} \setminus \{prog_{C_j}\} : \\
&\qquad\quad \widehat{wStart}^{UB}_{prog'_{C_j},C_j}(edg_{prog_{C_j}}) = \widehat{wStart}^{LB}_{prog'_{C_j},C_j}(edg_{prog_{C_j}}) = 0
\end{aligned} \tag{10.83}$$

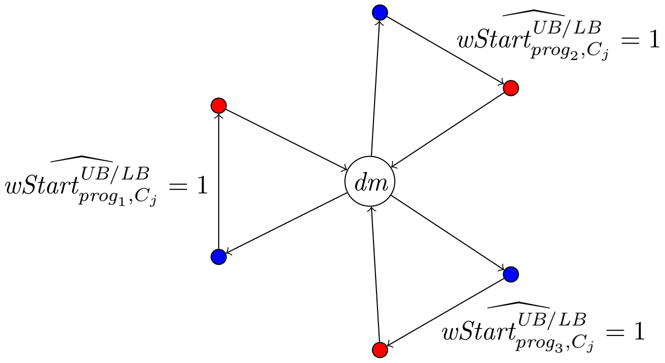

$\widehat{wStart}^{UB/LB}_{prog_2,C_j} = 1$
$\widehat{wStart}^{UB/LB}_{prog_1,C_j} = 1$
dm
$\widehat{wStart}^{UB/LB}_{prog_3,C_j} = 1$

Figure 10.13.: Generic scheduling graph for a scenario in which the considered processor core C_j only executes the example programs $prog_1$, $prog_2$, and $prog_3$ (i.e. $Programs_{C_j} = \{prog_1, prog_2, prog_3\}$). The figure only explicitly presents edge weights for program start events in case they have a non-zero value. Start nodes of the graph are colored in blue and end nodes are colored in red.

Additionally, the generic scheduling graph shall feature a single dummy node dm that has an outgoing edge to every program-specific source node $nd_{sc,prog_{C_j}}$ and an incoming edge from every program-specific target node $nd_{tg,prog_{C_j}}$. The incoming and outgoing edges of dummy node dm shall not describe any occurrences of a start event of a program executed on core C_j.

$$\begin{aligned}&\forall edg_{dm} \in \{(dm, nd_{sc,prog_{C_j}}), (nd_{tg,prog_{C_j}}, dm) \mid prog_{C_j} \in Programs_{C_j}\} :\\ &\quad \forall prog_{C_j} \in Programs_{C_j} :\\ &\quad\quad \widehat{wStart}^{UB}_{prog_{C_j},C_j}(edg_{dm}) = \widehat{wStart}^{LB}_{prog_{C_j},C_j}(edg_{dm}) = 0\end{aligned} \tag{10.84}$$

The set of start nodes of the generic scheduling graph shall be defined as the set of all program-specific source nodes. Analogously, the set of end nodes of the generic scheduling graph shall be defined as the set of all program-specific target nodes. Consequently, the generic scheduling graph G^{gs} shall be formally defined as follows.

$$G^{gs} = (Nodes^{gs}, Nodes^{gs}_{start}, Nodes^{gs}_{end}, Edges^{gs}) \tag{10.85}$$

$$Nodes^{gs} = \{dm, nd_{sc,prog_{C_j}}, nd_{tg,prog_{C_j}} \mid prog_{C_j} \in Programs_{C_j}\} \tag{10.86}$$

$$Nodes^{gs}_{start} = \{nd_{sc,prog_{C_j}} \mid prog_{C_j} \in Programs_{C_j}\} \tag{10.87}$$

$$Nodes^{gs}_{end} = \{nd_{tg,prog_{C_j}} \mid prog_{C_j} \in Programs_{C_j}\} \tag{10.88}$$

$$\begin{aligned}Edges^{gs} = \{&(nd_{sc,prog_{C_j}}, nd_{tg,prog_{C_j}}), (dm, nd_{sc,prog_{C_j}}), (nd_{tg,prog_{C_j}}, dm)\\ &\mid prog_{C_j} \in Programs_{C_j}\}\end{aligned} \tag{10.89}$$

This principle is depicted in Figure 10.13 for a scenario in which the considered processor core C_j only executes the example programs $prog_1$, $prog_2$, and $prog_3$ (i.e. $Programs_{C_j} = \{prog_1, prog_2, prog_3\}$). For the sake of readability, Figure 10.13 only explicitly presents edge weights for program start events in case they have a non-zero value.

Note that the idea behind the generic scheduling graph is very similar to directly considering multiple programs on the same processor core during implicit subpath enumeration (cf. Section 10.2.5). The graph on which the implicit subpath enumeration is performed features one relatively fine-grained sub graph per program executed on the considered processor core (cf. Figure 10.10). The generic scheduling graph, in contrast, only features one edge per program executed on the considered processor core (cf. Figure 10.13).

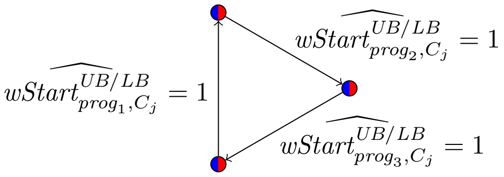

$\widehat{wStart}^{UB/LB}_{prog_2,C_j} = 1$
$\widehat{wStart}^{UB/LB}_{prog_1,C_j} = 1$
$\widehat{wStart}^{UB/LB}_{prog_3,C_j} = 1$

Figure 10.14.: Specialized scheduling graph for a scenario in which the considered processor core C_j executes the example programs $prog_1$, $prog_2$, and $prog_3$ in a cyclic fashion. The figure only explicitly presents edge weights for program start events in case they have a non-zero value. Start nodes of the graph are colored in blue and end nodes are colored in red.

The generic scheduling graph inherently assumes that the scheduler might execute the programs in any order. Most real-world scheduling scenarios for hard real-time embedded systems, however, are far more restricted. Thus, in such scenarios, there is typically more knowledge about the possible orders in which the programs can be executed. Such knowledge is expressed as a set of *scheduling properties* (i.e. system properties that argue about the amount or possible orders of occurrences of program start events in concrete traces).

The concept of graph transformations (cf. Section 6.4.6) can be used to blow up (cf. equation (6.C12)) the generic scheduling graph and to subsequently prune nodes and/or edges of the blown-up graph that are infeasible (cf. equation (6.C13)) according to lifted versions of the scheduling properties. As an example, consider a scheduling scenario in which the programs $prog_1$, $prog_2$, and $prog_3$ are executed by the scheduler only in the given order and in a cyclic manner (i.e. after $prog_3$, $prog_1$ is executed again). For this example scenario, the corresponding generic scheduling graph (cf. Figure 10.13) provides a very pessimistic overapproximation of the possible sequences of program execution runs as it describes many sequences of program execution runs that are actually infeasible with respect to the concrete traces. The specialized scheduling graph in Figure 10.14, in contrast, provides a precise overapproximation of the possible sequences of program execution runs. The specialized scheduling graph can be seen as the result of applying three graph transformations to the generic scheduling graph: First, a transformation blows up the graph by creating a specialized variant dm_1 (dm_2, dm_3) of the dummy node dm representing the case that $prog_1$ ($prog_2$, $prog_3$) was just executed (cf. equation (6.C12)). Subsequently, infeasible edges are pruned (e.g. the edges from dm_1 to $nd_{sc,prog_1}$ and $nd_{sc,prog_3}$, cf. equation (6.C13)). Finally, a transformation removes chains of edges along which all edge weights for the program start events are zero (cf. equation (6.C12)).

The following subsections demonstrate the use of a scheduling graph in a modular method for calculating arrival curve values. Intuitively, the precision of the calculated arrival curve values is often increased if the modular calculation is based on a graph that models the scheduling behavior of the considered processor core in a precise manner. In order to support this intuition with a practical example, reconsider a scenario in which the programs $prog_1$, $prog_2$, and $prog_3$ are scheduled in a non-preemptive way on processor core C_j. Further assume that the occurrences of event E in program $prog_1$ are significantly more dense than in the other two programs. Thus, a calculation of an arrival curve value for event E based on the generic scheduling graph (cf. Figure 10.13) would necessarily be dominated by a path through the graph that only describes execution runs of program $prog_1$. In case the programs executed on processor core C_j are scheduled in a cyclic manner, however, this dominating path through the generic scheduling graph is infeasible with respect to the concrete traces. Consequently, the arrival curve value calculated based on the generic scheduling graph is overly pessimistic and a calculation based on a more specialized scheduling graph (cf. Figure 10.14) is advisable.

Note that, for simplicity, the calculation of arrival curve values that we base on scheduling graphs (cf. Section 10.4.3 and Section 10.4.4) only uses bounds on event $\mathit{Start}_{prog,C_j}$ to express that program *prog* is executed on core C_j. For the calculation of sound arrival curve values to be used in a co-runner-sensitive WCET analysis for program $prog_{C_i}$ executed on core C_i, however, we also have to take into account execution runs of programs *prog* on core C_j that start before program $prog_{C_i}$ starts on core C_i but still overlap with the execution run of program $prog_{C_i}$ on core C_i. Thus, to be formally correct, the scheduling properties lifted to the level of paths through the scheduling graph must hold for all members of the following alternative set $\mathit{ExecRuns}^{sched}_{prog_{C_i},C_i}$ of concrete traces—instead of only for all members of $\mathit{ExecRuns}_{prog_{C_i},C_i}$. The definition of relation *SuffixOf* that we use to define the set shall be analogous to the definition of relation *PrefixOf* (cf. equation (6.25) on page 70).

$$\begin{aligned}\mathit{ExecRuns}^{sched}_{prog_{C_i},C_i} = \Big\{ t \in \mathit{Sequences} \mid \\ &\exists t' \in \mathit{ExecRuns}_{prog_{C_i},C_i} : \\ &\quad (t',t) \in \mathit{SuffixOf} \wedge \\ &\quad \sum_{x \in \mathbb{N}_{\leq len(t)-len(t')}} \sum_{prog \in \mathit{Programs}_{C_j}} \mathit{Start}_{prog,C_j}(t,x) \\ &\qquad \leq \max\{ \sum_{x \in \mathbb{N}_{<len(t'')}} \sum_{prog \in \mathit{Programs}_{C_j}} \mathit{End}_{prog,C_j}(t'',x) \mid \\ &\qquad\quad t'' \in \mathit{Sequences} \wedge (t'',t') \in \mathit{PrefixOf} \wedge \\ &\qquad\quad \sum_{x \in \mathbb{N}_{<len(t'')-1}} \sum_{prog \in \mathit{Programs}_{C_j}} \mathit{Start}_{prog,C_j}(t'',x) = 0\} \Big\}\end{aligned} \tag{10.90}$$

Consequently, the set $\mathit{Prop}^{sched}_{prog_{C_i},C_i}$ of scheduling properties that we lift to paths through the scheduling graph shall fulfill the following assumption.

$$\forall P_k \in \mathit{Prop}^{sched}_{prog_{C_i},C_i} : \forall t \in \mathit{ExecRuns}^{sched}_{prog_{C_i},C_i} : P_k(t) \tag{10.91}$$

The following subsections argue about a scheduling graph G^{sched} which coincides with the generic scheduling graph G^{gs} or has been derived from the generic scheduling graph G^{gs} by applying a sequence of graph transformations (cf. Section 6.4.6).

10.4.2. Implicit Scheduling Path Enumeration

The modular calculation of arrival curve values that we present in the following subsections does not depend on the order in which the program execution runs appear in a path through the scheduling graph G^{sched}. Thus, we apply the principle of implicit path enumeration (cf. Section 5.4) and, consequently, only consider implicit paths through the scheduling graph G^{sched}. We refer to this approach as *implicit scheduling path enumeration.* It argues about the set $\widehat{\mathit{Implicit}}^{sched}$ through the scheduling graph G^{sched}.

At the level of approximation of implicit scheduling path enumeration, we can optionally also use lifted versions of scheduling properties in order to prune infeasible implicit scheduling paths. The curve value calculation presented in the following subsection argues about the potentially reduced set $\widehat{\mathit{LessImplicit}}^{sched}$.

$$\widehat{\mathit{LessImplicit}}^{sched} = \{(tt, is, ie) \in \widehat{\mathit{Implicit}}^{sched} \mid \forall P_k \in \mathit{Prop}^{sched}_{prog_{C_i},C_i} : \widehat{P_k^{impli}}((tt, is, ie))\} \tag{10.92}$$

For some scheduling properties (as e.g. a property guaranteeing the cyclic nature of a scheduling), it is straight forward to hard-wire a lifted version of the property into a specialized scheduling graph (cf. Figure 10.14). For other scheduling properties, it is not practical to hard-wire a lifted version into the scheduling graph. As an example, consider a scheduling property stating that the number of times program $prog_1$ is executed must not exceed five plus three times the number of executions of program $prog_2$ plus two times the number of executions of program $prog_3$. It is not completely straight forward to hard-wire a lifted version of the example property into a specialized scheduling graph in a way that the graph precisely models the example property. In contrast, it is straight forward to precisely lift the example property to the level of approximation of implicit scheduling path enumeration as follows.

$$\begin{aligned}\widehat{P^{impli}_{xmpl}}((tt, is, ie)) \Leftrightarrow \sum_{edg \in Edges^{sched}} & tt(edg) \cdot \widehat{wStart^{LB}_{prog_1,C_j}}(edg) \\ \leq 5 + \sum_{edg \in Edges^{sched}} & tt(edg) \cdot \Bigg(3 \cdot \widehat{wStart^{UB}_{prog_2,C_j}}(edg) \\ & + 2 \cdot \widehat{wStart^{UB}_{prog_3,C_j}}(edg)\Bigg)\end{aligned} \tag{10.93}$$

For simplicity, the following subsections are based on an implicit scheduling path enumeration with binary variables. We expect that it is straight forward to come up with a variant of the presented calculation method that is based on an implicit scheduling path enumeration without binary variables (cf. Section 9.4). A more detailed discussion of such a variant, however, is beyond the scope of this thesis.

10.4.3. Curve Values Calculated at the Granularity of Program Runs

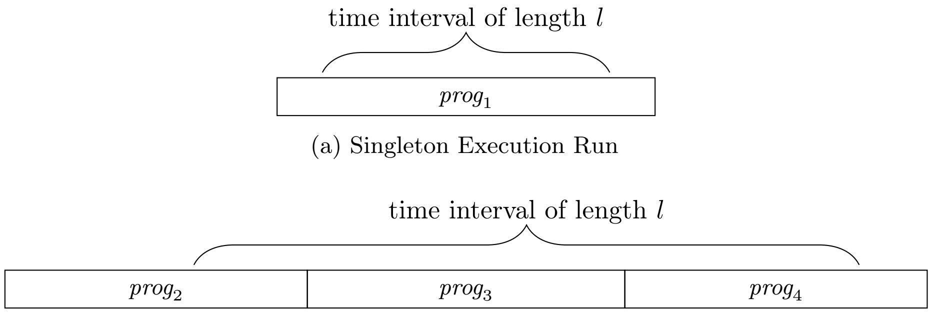
In this subsection, we demonstrate how to use implicit scheduling path enumeration for the calculation of arrival curve values at the granularity of program runs. To this end, we extend the implicit scheduling path enumeration by additional variables and constraints. As a starting point, we introduce an additional variable $Runs_{prog}$ per program $prog$ representing the number of execution runs of $prog$ on core C_j that is spanned by the considered implicit scheduling path.

$$\forall prog \in Programs_{C_j} : Runs_{prog} \in \mathbb{N} \tag{10.94}$$

The number of execution runs of program $prog$ is bounded from above and from below by the sums of the corresponding event-bounding edge weights for the program start event along the implicit scheduling path.

$$\begin{aligned}&\forall prog \in Programs_{C_j} : \\ &\qquad \sum_{edg \in Edges^{sched}} tt(edg) \cdot \widehat{wStart^{LB}_{prog,C_j}}(edg) \\ &\qquad \leq Runs_{prog} \\ &\qquad \leq \sum_{edg \in Edges^{sched}} tt(edg) \cdot \widehat{wStart^{UB}_{prog,C_j}}(edg)\end{aligned} \tag{10.95}$$

Each considered execution run of program $prog$ shall belong to exactly one of four possible categories of execution runs. In case an implicit scheduling path only describes one execution run of a single program, this execution run is considered as a *singleton execution run.* In case there are at least two execution runs of (potentially but not necessarily different) programs along the implicit scheduling path, there shall be exactly one *start execution run* (executed first along



(a) Singleton Execution Run

(b) Start Execution Run, Pass-Through Execution Run, and End Execution Run

Figure 10.15.: The four possible categories of execution runs considered during the calculation of arrival curve values. In case the considered time interval of length l only spans one execution run of a single program, this execution run is considered as a singleton execution run. In case the considered time interval spans at least two program execution runs, there is exactly one start execution run (executed first during the time interval), exactly one end execution run (executed last during the time interval), and an optional number of pass-through execution runs (executed in-between).

the implicit scheduling path), exactly one *end execution run* (executed last along the implicit scheduling path), and an optional number of *pass-through execution runs* (executed after the start run and before the end run).

$$\begin{aligned}&\forall prog \in Programs_{C_j} :\\ &\quad Runs_{prog} = SingletonRun_{prog} + StartRun_{prog} + EndRun_{prog} \\ &\qquad\qquad + PassThroughRuns_{prog}\end{aligned} \tag{10.96}$$

$$\forall prog \in Programs_{C_j} : SingletonRun_{prog} \in \{0,1\} \tag{10.97}$$

$$\forall prog \in Programs_{C_j} : StartRun_{prog} \in \{0,1\} \tag{10.98}$$

$$\forall prog \in Programs_{C_j} : EndRun_{prog} \in \{0,1\} \tag{10.99}$$

$$\forall prog \in Programs_{C_j} : PassThroughRuns_{prog} \in \mathbb{N} \tag{10.100}$$

The four possible categories of execution runs that are considered during the calculation of arrival curve values are depicted in Figure 10.15. Note that a time interval of length l that spans a sequence of execution runs described by an implicit scheduling path does not necessarily fully cover the first and the last execution run in the sequence. Thus, only the time spent during the execution of the spanned pass-through runs is guaranteed to be fully covered by the interval of length l.

In the following, we present some consistency constraints on the occurrences of the different categories of execution runs. To begin with, we require that an execution run of program *prog* is considered as a singleton execution run if and only if program *prog* performs a single execution run and all other programs perform no execution run at all along the implicit scheduling path.

$$\begin{aligned}&\forall prog \in Programs_{C_j} :\\ &\quad SingletonRun_{prog} \Leftrightarrow \Big[Runs_{prog} = 1 \\ &\qquad\qquad \wedge \forall prog' \in Programs_{C_j} \setminus \{prog\} : Runs_{prog'} = 0\Big]\end{aligned} \tag{10.101}$$

Note that the equivalence in equation (10.101) cannot directly be expressed in an ILP formulation. Thus, we simulate it by the following set of linear constraints making use of the two binary helper variables helper_1 and helper_2. For a formal sketch arguing that this set of linear constraints precisely simulates the equivalence in equation (10.101), we refer to page 315.

$$\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs}_{C_j} : \\
&\quad (1 - \mathit{SingletonRun}_{\mathit{prog}}) \cdot l \geq \sum_{\mathit{prog}' \in \mathit{Programs}_{C_j} \setminus \{\mathit{prog}\}} \mathit{Runs}_{\mathit{prog}'} && (10.102)\\
&\mathit{helper}_1 \cdot l \geq \sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{Runs}_{\mathit{prog}} && (10.103)\\
&\sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{Runs}_{\mathit{prog}} + \mathit{helper}_2 \cdot 2 \geq 2 && (10.104)\\
&1 + \sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{SingletonRun}_{\mathit{prog}} \geq \mathit{helper}_1 + \mathit{helper}_2 && (10.105)
\end{aligned}$$

It follows from the equivalence in equation (10.101) that there can be at most one singleton execution run over all programs executed on core C_j. We exploit this to express that every implicit scheduling path shall have exactly one start execution run and exactly one end execution run in case it has no singleton execution run. Analogously, every implicit scheduling path shall have no start execution run and no end execution run in case it has a singleton execution run.

$$\begin{aligned}
&\sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{StartRun}_{\mathit{prog}} \\
&= \sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{EndRun}_{\mathit{prog}} \qquad (10.106)\\
&= 1 - \sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{SingletonRun}_{\mathit{prog}}
\end{aligned}$$

Moreover, we require that a program *prog* may only have a singleton execution run or a start execution run in case the implicit scheduling path begins with an edge that permits the occurrence of a start event of program *prog*. Analogously, a program *prog* may only have a singleton execution run or an end execution run in case the implicit scheduling path ends with an edge that permits the occurrence of a start event of program *prog*.

$$\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs}_{C_j} : \\
&\quad \mathit{SingletonRun}_{\mathit{prog}} + \mathit{StartRun}_{\mathit{prog}} \leq \sum_{\mathit{edg} \in \mathit{Edges}^{\mathit{sched}}} \mathit{is}(\mathit{edg}) \cdot \widehat{\mathit{wStart}}^{UB}_{\mathit{prog},C_j}(\mathit{edg}) && (10.107)\\
&\forall \mathit{prog} \in \mathit{Programs}_{C_j} : \\
&\quad \mathit{SingletonRun}_{\mathit{prog}} + \mathit{EndRun}_{\mathit{prog}} \leq \sum_{\mathit{edg} \in \mathit{Edges}^{\mathit{sched}}} \mathit{ie}(\mathit{edg}) \cdot \widehat{\mathit{wStart}}^{UB}_{\mathit{prog},C_j}(\mathit{edg}) && (10.108)
\end{aligned}$$

Next, we distribute time in clock cycles across the different categories of execution runs of the different programs. To this end, we introduce one time variable per program and category of execution run. The sum of the time variables of all programs shall coincide with the interval length l.

$$\forall prog \in Programs_{C_j} : SingletonTime_{prog} \in \mathbb{N} \tag{10.109}$$

$$\forall prog \in Programs_{C_j} : StartTime_{prog} \in \mathbb{N} \tag{10.110}$$

$$\forall prog \in Programs_{C_j} : EndTime_{prog} \in \mathbb{N} \tag{10.111}$$

$$\forall prog \in Programs_{C_j} : PassThroughTime_{prog} \in \mathbb{N} \tag{10.112}$$

$$\begin{aligned} \sum_{prog \in Programs_{C_j}} \big[& SingletonTime_{prog} + StartTime_{prog} \\ & + EndTime_{prog} + PassThroughTime_{prog} \big] = l \end{aligned} \tag{10.113}$$

The time that any program spends within an execution run of any category shall range from one clock cycle to the interval length l. Note that a program shall not spend any time within an execution run of a particular category if it is not supposed to perform an execution run of this category according to the current implicit scheduling path.

$$\begin{gathered} \forall prog \in Programs_{C_j} : \\ SingletonRun_{prog} \cdot 1 \leq SingletonTime_{prog} \leq SingletonRun_{prog} \cdot l \end{gathered} \tag{10.114}$$

$$\begin{gathered} \forall prog \in Programs_{C_j} : \\ StartRun_{prog} \cdot 1 \leq StartTime_{prog} \leq StartRun_{prog} \cdot l \end{gathered} \tag{10.115}$$

$$\begin{gathered} \forall prog \in Programs_{C_j} : \\ EndRun_{prog} \cdot 1 \leq EndTime_{prog} \leq EndRun_{prog} \cdot l \end{gathered} \tag{10.116}$$

$$\begin{gathered} \forall prog \in Programs_{C_j} : \\ PassThroughRuns_{prog} \cdot 1 \leq PassThroughTime_{prog} \leq \min(PassThroughRuns_{prog} \cdot l,\ l) \end{gathered} \tag{10.117}$$

In case there is a BCET bound for a program *prog* available, we can use it as an additional lower bound for the amount of time that program *prog* has to spend in any pass-through execution run. Intuitively, any pass-through execution run of program *prog* is executed from the program start to the program end of *prog* (cf. Figure 10.15b). Thus, any pass-through execution run of program *prog* takes at least the BCET of program *prog*, which is safely under-approximated by a BCET bound.

$$PassThroughRuns_{prog} \cdot BCET^{LB}_{prog,C_j} \leq PassThroughTime_{prog} \tag{10.118}$$

Analogously, in case there is a minimum inter-start time enforced for a program *prog*, we can use it as an additional lower bound for the amount of time that program *prog* has to spend in any pass-through execution run. Intuitively, any pass-through execution run of program *prog* is executed from the program start to the program end of *prog* and also accounts for potentially added clock cycles after the execution run due to the enforcement of the minimum inter-start time. Thus, any pass-through execution run of program *prog* takes at least the minimum inter-start time of program *prog*.

$$PassThroughRuns_{prog} \cdot MIST_{prog} \leq PassThroughTime_{prog} \tag{10.119}$$

Next, we distribute occurrences of event E across the different categories of execution runs of the different programs. To this end, we introduce one event counter variable per program and category of execution run.

$$\forall prog \in Programs_{C_j} : SingletonOccs_{prog,E} \in \mathbb{N} \tag{10.120}$$

$$\forall prog \in Programs_{C_j} : StartOccs_{prog,E} \in \mathbb{N} \tag{10.121}$$

$$\forall prog \in Programs_{C_j} : EndOccs_{prog,E} \in \mathbb{N} \tag{10.122}$$

$$\forall prog \in Programs_{C_j} : PassThroughOccs_{prog,E} \in \mathbb{N} \tag{10.123}$$

In our event model, any clock cycle can at most produce one occurrence of any event E (cf. equation (5.6)). This is reflected by the following constraints that bound the number of event occurrences in an execution run by the amount of clock cycles spent in the execution run.

$$\forall prog \in Programs_{C_j} : SingletonOccs_{prog,E} \leq SingletonTime_{prog} \cdot 1 \tag{10.124}$$

$$\forall prog \in Programs_{C_j} : StartOccs_{prog,E} \leq StartTime_{prog} \cdot 1 \tag{10.125}$$

$$\forall prog \in Programs_{C_j} : EndOccs_{prog,E} \leq EndTime_{prog} \cdot 1 \tag{10.126}$$

$$\forall prog \in Programs_{C_j} : PassThroughOccs_{prog,E} \leq PassThroughTime_{prog} \cdot 1 \tag{10.127}$$

For a given event E, we might be able to provide significantly more precise upper bounds on the number of event occurrences. We demonstrate this for the example of event $Completed_{C_j}$, which occurs when a bus access of core C_j completes (cf. Section 7.2). The share of interval length l that is spent during the execution of a singleton run or a start run of program *prog* does not necessarily have to begin at the program start of *prog* (cf. Figure 10.15). Consequently, one clock cycle spent in such an execution run may already exhibit an occurrence of event $Completed_{C_j}$. Any further occurrence of event $Completed_{C_j}$, however, requires at least LAT more clock cycles as any bus access takes at least the latency LAT until completion. This is reflected by the following constraints bounding the number of occurrences of event $Completed_{C_j}$ in singleton and start execution runs.

$$\begin{aligned}&\forall prog \in Programs_{C_j} :\\ &\quad SingletonOccs_{prog,Completed_{C_j}} \leq \frac{LAT-1}{LAT} + SingletonTime_{prog} \cdot \frac{1}{LAT}\end{aligned} \tag{10.128}$$

$$\begin{aligned}&\forall prog \in Programs_{C_j} :\\ &\quad StartOccs_{prog,Completed_{C_j}} \leq \frac{LAT-1}{LAT} + StartTime_{prog} \cdot \frac{1}{LAT}\end{aligned} \tag{10.129}$$

The amount of time that is spent in an end execution run or a pass-through execution run of program *prog*, in contrast, is guaranteed to begin at the program start of *prog* (cf. Figure 10.15). Moreover, we enforce a software convention that guarantees that the pipeline of every processor core is drained whenever a program executed on the core reaches its program end (cf. end of Section 6.1). Thus, bus accesses of core C_j never span across multiple execution runs on core C_j. As a consequence, from the program start of an end execution run or a pass-through execution run of program *prog*, it takes at least LAT clock cycles until the first bus access completes. This is reflected by the following more precise constraints bounding the number of occurrences of event $Completed_{C_j}$ in end and pass-through execution runs.

$$\begin{aligned}&\forall prog \in Programs_{C_j} :\\ &\quad EndOccs_{prog,Completed_{C_j}} \leq EndTime_{prog} \cdot \frac{1}{LAT}\end{aligned} \tag{10.130}$$

$$\begin{aligned}&\forall prog \in Programs_{C_j} :\\ &\quad PassThroughOccs_{prog,Completed_{C_j}} \leq PassThroughTime_{prog} \cdot \frac{1}{LAT}\end{aligned} \tag{10.131}$$

In case there is a per-execution-run event bound $Max^{UB}_{prog,C_j,E}$ for a program *prog* available (cf. Chapter 6), we can use it as an additional upper bound for the number of occurrences of event E in any execution run of program *prog*.

$$SingletonOccs_{prog,E} \leq SingletonRun_{prog} \cdot Max^{UB}_{prog,C_j,E} \tag{10.132}$$

$$StartOccs_{prog,E} \leq StartRun_{prog} \cdot Max^{UB}_{prog,C_j,E} \tag{10.133}$$

$$EndOccs_{prog,E} \leq EndRun_{prog} \cdot Max^{UB}_{prog,C_j,E} \tag{10.134}$$

$$PassThroughOccs_{prog,E} \leq PassThroughRuns_{prog} \cdot Max^{UB}_{prog,C_j,E} \tag{10.135}$$

Finally, we calculate an arrival curve value upper-bounding the number of occurrences of event E in any time interval of l clock cycles by maximizing the following objective.

$$\begin{aligned}\max \sum_{prog \in Programs_{C_j}} \big[& SingletonOccs_{prog,E} + StartOccs_{prog,E} \\ & + EndOccs_{prog,E} + PassThroughOccs_{prog,E} \big]\end{aligned} \tag{10.136}$$

Note that this calculation of an arrival curve value *operates at the granularity of program runs* (cf. Section 10.1) as it inherently assumes that the maximal amount of $Max^{UB}_{prog,C_j,E}$ event occurrences that an execution run of program *prog* generates can be distributed across the execution run in an arbitrary fashion. In the following subsection, we present an extension of this calculation that operates at a finer granularity by additionally incorporating compositional base bounds (cf. Chapter 8) for program *prog* that are calculated based on an implicit subpath enumeration (cf. Section 10.2) of a graph representation of program *prog*.

10.4.4. Beyond the Granularity of Program Runs

In this subsection, we demonstrate how to additionally bound the number of occurrences of event E in the execution runs of a program *prog* in a more fine-grained way than at the granularity of program runs. To this end, we calculate a compositional base bound for program *prog* bounding the number of occurrences of event E in any execution run of *prog* under the assumption that every clock cycle spent in the execution run contributes a given and fixed amount of *pen* ($\in \mathbb{R}_{\geq 0} \cap \mathbb{R}_{\leq 1}$) occurrences of event E.

As a starting point, each of the following calculations of compositional base bounds relies on a graph $G^{C,prog}$ as used for the optimized calculation of per-execution-run compositional base bounds for program *prog* executed on core C_j ($CompBaseBounds_{prog,C_j}(E, \leq, \{(pen, Cycle)\})$, cf. Section 8.3).

In the following, we present four calculations of compositional base bounds. The compositional base bounds resulting from the first calculation are valid for all categories of execution runs. The other three calculations result in potentially more precise compositional base bounds for start execution runs, end execution runs, respectively pass-through execution runs.

A Compositional Base Bound Valid for All Categories of Execution Runs The first calculation of compositional base bounds results in bounds that are valid for all categories of execution runs. In particular, the resulting bounds have to be valid for singleton execution runs, in which the time interval considered during the calculation of arrival curve values may begin at an arbitrary point and end at an arbitrary later point of an execution run of program *prog* on core C_j (cf. Figure 10.15a). This means that the bound calculation has to argue about all possible *sub-traces of concrete traces* occurring when program *prog* is executed on core C_j.

$$ExecRuns^{sub}_{prog,C_j} = \{t \in Sequences \mid \exists t' \in ExecRuns_{prog,C_j} : (t, t') \in SuffixOf\} \tag{10.137}$$

Consequently, the first bound calculation argues about all subpaths of graph $G^{C,prog}$. To this end, it operates on a graph $G^{C,prog,sub}$ that is obtained from graph $G^{C,prog}$ by marking each node as start node and as end node.

$$Nodes^{C,prog,sub} = Nodes^{C,prog} \tag{10.138}$$
$$Nodes^{C,prog,sub}_{start} = Nodes^{C,prog} \tag{10.139}$$
$$Nodes^{C,prog,sub}_{end} = Nodes^{C,prog} \tag{10.140}$$
$$Edges^{C,prog,sub} = Edges^{C,prog} \tag{10.141}$$

We assume that there is a set $Prop^{sub}_{prog,C_j}$ of system properties that hold for all concrete traces of set $ExecRuns^{sub}_{prog,C_j}$. Note that this set also contains the special flavor of loop-bounding properties for the calculation of arrival curve values that we use in Section 10.2 (cf. equation (10.20)).

$$\forall P_k \in Prop^{sub}_{prog,C_j} : \forall t \in ExecRuns^{sub}_{prog,C_j} : P_k(t) \tag{10.142}$$

Lifted versions of the system properties in set $Prop^{sub}_{prog,C_j}$ are used for pruning infeasible implicit paths through graph $G^{C,prog,sub}$.

$$\begin{aligned}\widehat{LessImplicit}^{C,prog,sub} = \{&(tt, is, ie) \in \widehat{Implicit}^{C,prog,sub} \\ &\mid \forall P_k \in Prop^{sub}_{prog,C_j} : \widehat{P^{impli}_k}((tt, is, ie))\}\end{aligned} \tag{10.143}$$

The actual calculation of a compositional base bound is performed on the remaining set of implicit paths as follows. Note, however, that a discussion of the soundness of this calculation is omitted due to time and space constraints.

$$\begin{aligned}CompBase^{sub}_{prog,C_j,E,pen} = \max_{(tt,is,ie)\in \widehat{LessImplicit}^{C,prog,sub}} \sum_{edg\in Edges^{C,prog,sub}} &[tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_j,Cmp}(edg) \\ &- is(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_j,Cmp}(edg) \\ &+ is(edg) \cdot (1 - pen) \cdot \widehat{wEvent}^{UB}_{prog,C_j,E}(edg) \\ &- ie(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_j,Cmp}(edg) \\ &+ ie(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_j,E}(edg)]\end{aligned} \tag{10.144}$$

In case graph $G^{C,prog}$ is at the granularity of cycle transitions (cf. Section 6.4.2), it is safe to simplify the calculation of the compositional base bound as follows.

$$\begin{aligned}CompBase^{sub}_{prog,C_j,E,pen} = \max_{(tt,is,ie)\in \widehat{LessImplicit}^{C,prog,sub}} \sum_{edg\in Edges^{C,prog,sub}} \\ tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_j,Cmp}(edg)\end{aligned} \tag{10.145}$$

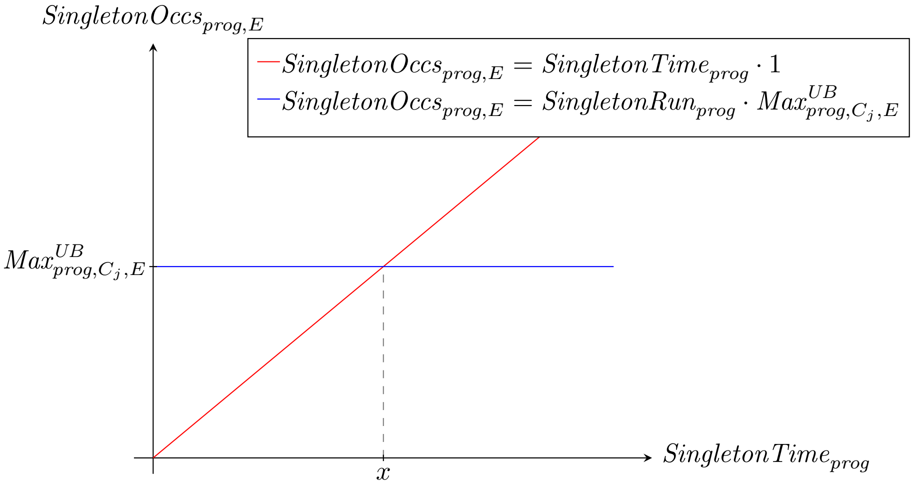


Figure 10.16.: The number of occurrences of event E in a singleton execution run of program *prog* ($SingletonOccs_{prog,E}$) as a function of the number of clock cycles spent in the singleton execution run during the time interval of length l ($SingletonTime_{prog}$) for the case that a singleton execution run of program *prog* takes place (i.e. $SingletonRun_{prog} = 1$). The value of $SingletonOccs_{prog,E}$ calculated in Section 10.4.3 is only bounded by two linear functions.

In case this compositional base bound has a defined value, the value is used in the following constraints that we add to the calculation of arrival curve values presented in Section 10.4.3.

$$\begin{aligned} SingletonOccs_{prog,E} \leq{}& SingletonRun_{prog} \cdot CompBase^{sub}_{prog,C_j,E,pen} \\ &+ SingletonTime_{prog} \cdot pen \end{aligned} \tag{10.146}$$

$$\begin{aligned} StartOccs_{prog,E} \leq{}& StartRun_{prog} \cdot CompBase^{sub}_{prog,C_j,E,pen} \\ &+ StartTime_{prog} \cdot pen \end{aligned} \tag{10.147}$$

$$\begin{aligned} EndOccs_{prog,E} \leq{}& EndRun_{prog} \cdot CompBase^{sub}_{prog,C_j,E,pen} \\ &+ EndTime_{prog} \cdot pen \end{aligned} \tag{10.148}$$

$$\begin{aligned} PassThroughOccs_{prog,E} \leq{}& PassThroughRuns_{prog} \cdot CompBase^{sub}_{prog,C_j,E,pen} \\ &+ PassThroughTime_{prog} \cdot pen \end{aligned} \tag{10.149}$$

In the following, we provide an intuition for the potential gain in precision due to the additional constraints featuring the compositional base bound. To this end, we consider the number of occurrences of event E in a singleton execution run of program *prog* ($SingletonOccs_{prog,E}$) as a function of the number of clock cycles spent in the singleton execution run during the time interval of length l ($SingletonTime_{prog}$) for the case that a singleton execution run of program *prog* takes place (i.e. $SingletonRun_{prog} = 1$). Figure 10.16 depicts the situation without constraint (10.146) as presented in Section 10.4.3. The red line has a slope of one and corresponds to constraint (10.124). The blue line has a slope of zero and corresponds to constraint (10.132). In case the value of $SingletonTime_{prog}$ is smaller than x, the red line results in a more precise value for $SingletonOccs_{prog,E}$. In case the value of $SingletonTime_{prog}$ is greater than x, the blue line results in a more precise value for $SingletonOccs_{prog,E}$.

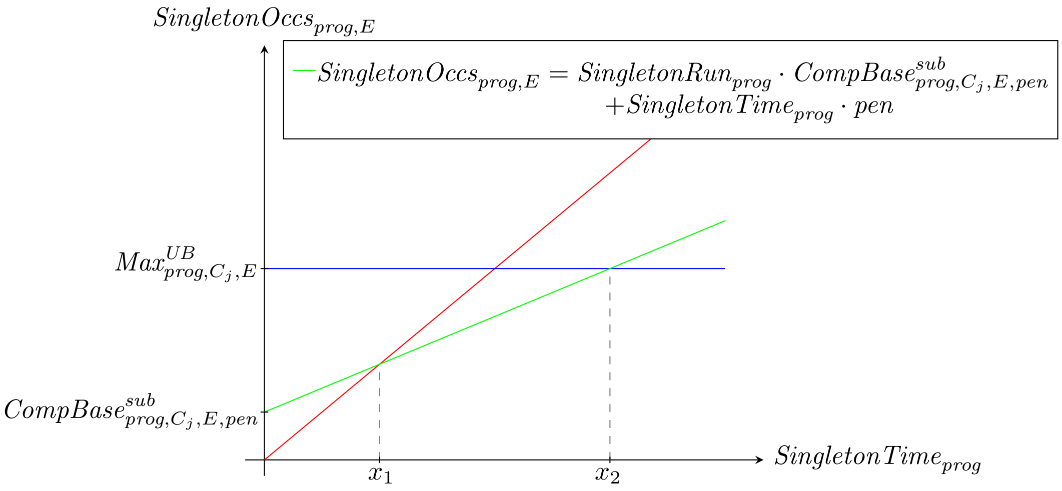


Figure 10.17.: The incorporation of an additional linear constraint improves the precision for values of $SingletonTime_{prog}$ between x_1 and x_2.

Figure 10.17 demonstrates the potential gain in precision by additionally incorporating constraint (10.146). The figure assumes that the slope *pen* is greater than zero and smaller than one. For a value of $SingletonTime_{prog}$ that is between x_1 and x_2, the additional constraint results in a more precise value of $SingletonOccs_{prog,E}$.

A Compositional Base Bound Valid for Start Execution Runs and Pass-Through Execution Runs The second calculation of compositional base bounds results in bounds that are valid for start execution runs and pass-through execution runs. In particular, the resulting bounds have to be valid for start execution runs, in which the time interval considered during the calculation of arrival curve values may begin at an arbitrary point of an execution run of program *prog* on core C_j (cf. Figure 10.15b). This means that the bound calculation has to argue about all possible *suffixes of terminated concrete traces* occurring when program *prog* is executed on core C_j.

$$ExecRuns^{suff}_{prog,C_j} = \{t \in Sequences \mid \exists t' \in ExecRuns^{term}_{prog,C_j} : (t,t') \in SuffixOf\} \tag{10.150}$$

Consequently, the second bound calculation argues about all suffixes of paths through graph $G^{C,prog}$. To this end, it operates on a graph $G^{C,prog,suff}$ that is obtained from graph $G^{C,prog}$ by marking each node as start node.

$$Nodes^{C,prog,suff} = Nodes^{C,prog} \tag{10.151}$$
$$Nodes^{C,prog,suff}_{start} = Nodes^{C,prog} \tag{10.152}$$
$$Nodes^{C,prog,suff}_{end} = Nodes^{C,prog}_{end} \tag{10.153}$$
$$Edges^{C,prog,suff} = Edges^{C,prog} \tag{10.154}$$

We assume that there is a set $Prop^{suff}_{prog,C_j}$ of system properties that hold for all concrete traces of set $ExecRuns^{suff}_{prog,C_j}$. Note that this set also contains the special flavor of loop-bounding properties for the calculation of arrival curve values that we use in Section 10.2 (cf. equation (10.20)).

$$\forall P_k \in Prop^{suff}_{prog,C_j} : \forall t \in ExecRuns^{suff}_{prog,C_j} : P_k(t) \tag{10.155}$$

Lifted versions of the system properties in set $\mathit{Prop}^{\mathit{suff}}_{\mathit{prog},C_j}$ are used for pruning infeasible implicit paths through graph $G^{C,\mathit{prog},\mathit{suff}}$.

$$\begin{aligned}\widehat{\mathit{LessImplicit}}^{C,\mathit{prog},\mathit{suff}} = \{(\mathit{tt}, \mathit{is}, \mathit{ie}) \in \widehat{\mathit{Implicit}}^{C,\mathit{prog},\mathit{suff}} \\ \mid \forall P_k \in \mathit{Prop}^{\mathit{suff}}_{\mathit{prog},C_j} : \widehat{P_k^{\mathit{impli}}}((\mathit{tt}, \mathit{is}, \mathit{ie}))\}\end{aligned} \tag{10.156}$$

The actual calculation of a compositional base bound is performed on the remaining set of implicit paths as follows. Note, however, that a discussion of the soundness of this calculation is omitted due to time and space constraints.

$$\begin{aligned}\mathit{CompBase}^{\mathit{suff}}_{\mathit{prog},C_j,E,\mathit{pen}} = \max_{(\mathit{tt},\mathit{is},\mathit{ie}) \in \widehat{\mathit{LessImplicit}}^{C,\mathit{prog},\mathit{suff}}} \sum_{\mathit{edg} \in \mathit{Edges}^{C,\mathit{prog},\mathit{suff}}} \\ [\mathit{tt}(\mathit{edg}) \cdot \widehat{\mathit{wEvent}}^{UB}_{\mathit{prog},C_j,\mathit{Cmp}}(\mathit{edg}) \\ - \mathit{is}(\mathit{edg}) \cdot \widehat{\mathit{wEvent}}^{UB}_{\mathit{prog},C_j,\mathit{Cmp}}(\mathit{edg}) \\ + \mathit{is}(\mathit{edg}) \cdot (1 - \mathit{pen}) \cdot \widehat{\mathit{wEvent}}^{UB}_{\mathit{prog},C_j,E}(\mathit{edg})]\end{aligned} \tag{10.157}$$

In case graph $G^{C,\mathit{prog}}$ is at the granularity of cycle transitions (cf. Section 6.4.2), it is safe to simplify the calculation of the compositional base bound as follows.

$$\begin{aligned}\mathit{CompBase}^{\mathit{suff}}_{\mathit{prog},C_j,E,\mathit{pen}} = \max_{(\mathit{tt},\mathit{is},\mathit{ie}) \in \widehat{\mathit{LessImplicit}}^{C,\mathit{prog},\mathit{suff}}} \sum_{\mathit{edg} \in \mathit{Edges}^{C,\mathit{prog},\mathit{suff}}} \\ \mathit{tt}(\mathit{edg}) \cdot \widehat{\mathit{wEvent}}^{UB}_{\mathit{prog},C_j,\mathit{Cmp}}(\mathit{edg})\end{aligned} \tag{10.158}$$

In case this compositional base bound has a defined value, the value is used in the following constraints that we add to the calculation of arrival curve values presented in Section 10.4.3.

$$\begin{aligned}\mathit{StartOccs}_{\mathit{prog},E} \leq \mathit{StartRun}_{\mathit{prog}} \cdot \mathit{CompBase}^{\mathit{suff}}_{\mathit{prog},C_j,E,\mathit{pen}} \\ + \mathit{StartTime}_{\mathit{prog}} \cdot \mathit{pen}\end{aligned} \tag{10.159}$$

$$\begin{aligned}\mathit{PassThroughOccs}_{\mathit{prog},E} \leq \mathit{PassThroughRuns}_{\mathit{prog}} \cdot \mathit{CompBase}^{\mathit{suff}}_{\mathit{prog},C_j,E,\mathit{pen}} \\ + \mathit{PassThroughTime}_{\mathit{prog}} \cdot \mathit{pen}\end{aligned} \tag{10.160}$$

A Compositional Base Bound Valid for End Execution Runs and Pass-Through Execution Runs The third calculation of compositional base bounds results in bounds that are valid for end execution runs and pass-through execution runs. In particular, the resulting bounds have to be valid for end execution runs, in which the time interval considered during the calculation of arrival curve values may end at an arbitrary point of an execution run of program *prog* on core C_j (cf. Figure 10.15b). This means that the bound calculation has to argue about all possible *prefixes of concrete traces* occurring when program *prog* is executed on core C_j. Note that this set of prefixes is referred to as $\mathit{ExecRuns}_{\mathit{prog},C_j}$ (cf. corresponding definition on page 68).

Consequently, the third bound calculation argues about all prefixes of paths through graph $G^{C,\mathit{prog}}$. To this end, it operates on a graph $G^{C,\mathit{prog},\mathit{pre}}$ that is obtained from graph $G^{C,\mathit{prog}}$ by marking each node as end node.

$$\mathit{Nodes}^{C,\mathit{prog},\mathit{pre}} = \mathit{Nodes}^{C,\mathit{prog}} \tag{10.161}$$

$$\mathit{Nodes}^{C,\mathit{prog},\mathit{pre}}_{\mathit{start}} = \mathit{Nodes}^{C,\mathit{prog}}_{\mathit{start}} \tag{10.162}$$

$$\mathit{Nodes}^{C,\mathit{prog},\mathit{pre}}_{\mathit{end}} = \mathit{Nodes}^{C,\mathit{prog}} \tag{10.163}$$

$$\mathit{Edges}^{C,\mathit{prog},\mathit{pre}} = \mathit{Edges}^{C,\mathit{prog}} \tag{10.164}$$

Recall that there is a set Prop_{prog,C_j} of system properties that hold for all concrete traces of set $\mathit{ExecRuns}_{prog,C_j}$. Note that this set also contains the classical loop-bounding properties used for the calculation of per-execution-run event bounds (cf. equation (10.16)).

Lifted versions of the system properties in set Prop_{prog,C_j} are used for pruning infeasible implicit paths through graph $G^{C,prog,pre}$.

$$\begin{aligned}\widehat{\mathit{LessImplicit}}^{C,prog,pre} = \{&(tt, is, ie) \in \widehat{\mathit{Implicit}}^{C,prog,pre} \\ &\mid \forall P_k \in \mathit{Prop}_{prog,C_j} : \widehat{P_k^{impli}}((tt, is, ie))\}\end{aligned} \tag{10.165}$$

The actual calculation of a compositional base bound is performed on the remaining set of implicit paths as follows. Note, however, that a discussion of the soundness of this calculation is omitted due to time and space constraints.

$$\begin{aligned}\mathit{CompBase}^{pre}_{prog,C_j,E,pen} = \max_{(tt,is,ie)\in\widehat{\mathit{LessImplicit}}^{C,prog,pre}} \sum_{edg\in \mathit{Edges}^{C,prog,pre}} &[tt(edg) \cdot \widehat{\mathit{wEvent}}^{UB}_{prog,C_j,Cmp}(edg) \\ &- ie(edg) \cdot \widehat{\mathit{wEvent}}^{UB}_{prog,C_j,Cmp}(edg) \\ &+ ie(edg) \cdot \widehat{\mathit{wEvent}}^{UB}_{prog,C_j,E}(edg)]\end{aligned} \tag{10.166}$$

In case graph $G^{C,prog}$ is at the granularity of cycle transitions (cf. Section 6.4.2), it is safe to simplify the calculation of the compositional base bound as follows.

$$\begin{aligned}\mathit{CompBase}^{pre}_{prog,C_j,E,pen} = \max_{(tt,is,ie)\in\widehat{\mathit{LessImplicit}}^{C,prog,pre}} \sum_{edg\in \mathit{Edges}^{C,prog,pre}} \\ tt(edg) \cdot \widehat{\mathit{wEvent}}^{UB}_{prog,C_j,Cmp}(edg)\end{aligned} \tag{10.167}$$

In case this compositional base bound has a defined value, the value is used in the following constraints that we add to the calculation of arrival curve values presented in Section 10.4.3.

$$\begin{aligned}\mathit{EndOccs}_{prog,E} \leq\; &\mathit{EndRun}_{prog} \cdot \mathit{CompBase}^{pre}_{prog,C_j,E,pen} \\ &+ \mathit{EndTime}_{prog} \cdot pen\end{aligned} \tag{10.168}$$

$$\begin{aligned}\mathit{PassThroughOccs}_{prog,E} \leq\; &\mathit{PassThroughRuns}_{prog} \cdot \mathit{CompBase}^{pre}_{prog,C_j,E,pen} \\ &+ \mathit{PassThroughTime}_{prog} \cdot pen\end{aligned} \tag{10.169}$$

A Compositional Base Bound Valid for Pass-Through Execution Runs The fourth calculation of compositional base bounds results in bounds that are valid for pass-through execution runs. This means that the bound calculation has to argue about all possible *terminated concrete traces* occurring when program *prog* is executed on core C_j. Note that this set of concrete traces is referred to as $\mathit{ExecRuns}^{term}_{prog,C_j}$ (cf. corresponding definition on page 69). Consequently, the fourth bound calculation argues about all paths through graph $G^{C,prog}$.

We assume that there is a set $\mathit{Prop}^{term}_{prog,C_j}$ of system properties that hold for all concrete traces of set $\mathit{ExecRuns}^{term}_{prog,C_j}$. Note that this set also contains the classical loop-bounding properties used for the calculation of per-execution-run event bounds (cf. equation (10.16)).

$$\forall P_k \in \mathit{Prop}^{term}_{prog,C_j} : \forall t \in \mathit{ExecRuns}^{term}_{prog,C_j} : P_k(t) \tag{10.170}$$

Lifted versions of the system properties in set $Prop^{term}_{prog,C_j}$ are used for pruning infeasible implicit paths through graph $G^{C,prog}$.

$$\begin{aligned} \widehat{LessImplicit}^{C,prog} = \{(tt, is, ie) \in \widehat{Implicit}^{C,prog} \\ \mid \forall P_k \in Prop^{term}_{prog,C_j} : \widehat{P_k^{impli}}((tt, is, ie))\} \end{aligned} \tag{10.171}$$

The actual calculation of a compositional base bound is performed on the remaining set of implicit paths as follows. Note, however, that a discussion of the soundness of this calculation is omitted due to time and space constraints.

$$\begin{aligned} CompBase^{term}_{prog,C_j,E,pen} = \max_{(tt,is,ie)\in \widehat{LessImplicit}^{C,prog}} \sum_{edg\in Edges^{C,prog}} \\ tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_j,Cmp}(edg) \end{aligned} \tag{10.172}$$

In case this compositional base bound has a defined value, the value is used in the following constraint that we add to the calculation of arrival curve values presented in Section 10.4.3.

$$\begin{aligned} PassThroughOccs_{prog,E} \leq PassThroughRuns_{prog} \cdot CompBase^{term}_{prog,C_j,E,pen} \\ + PassThroughTime_{prog} \cdot pen \end{aligned} \tag{10.173}$$

Relative Comparison of the Expected Computational Complexities of the Four Calculations of Compositional Base Bounds The calculation of compositional base bound $CompBase^{sub}_{prog,C_j,E,pen}$ argues about all subpaths of graph $G^{C,prog}$. Thus, this calculation is considered as an *implicit subpath enumeration* (cf. Section 10.2.3). Analogously, the calculation of compositional base bound $CompBase^{suff}_{prog,C_j,E,pen}$ is considered as an *implicit suffix path enumeration* and the calculation of compositional base bound $CompBase^{pre}_{prog,C_j,E,pen}$ is considered as an *implicit prefix path enumeration.* The calculation of compositional base bound $CompBase^{term}_{prog,C_j,E,pen}$ argues about all paths through graph $G^{C,prog}$ and, thus, is considered as a classical *implicit path enumeration.*

The experiments in Section 10.2 indicate that the computational complexity of implicit subpath enumeration is significantly higher than the computational complexity of classical implicit path enumeration. This is due to the set of implicit paths considered during implicit subpath enumeration typically being significantly larger than the set of implicit paths considered during classical implicit path enumeration. Note that, in particular, implicit subpath enumeration considers each implicit path that is considered during classical implicit path enumeration.

Consequently, we expect that the computational complexity of the calculation of compositional base bound $CompBase^{sub}_{prog,C_j,E,pen}$ is significantly higher than the computational complexity of the calculation of compositional base bound $CompBase^{term}_{prog,C_j,E,pen}$. Correspondingly, we expect the computational complexities of the calculations of $CompBase^{suff}_{prog,C_j,E,pen}$ and $CompBase^{pre}_{prog,C_j,E,pen}$ to be in between the computational complexities of aforementioned calculations. The expected computational complexities of the four calculations and the corresponding subset relations between the considered sets of implicit paths are sketched in Figure 10.18.

10.4.5. Advantages of the Sketched Program-Modular Calculation Method

In this subsection, we discuss the advantages of the sketched program-modular calculation method for values on arrival curves compared to the non-modular calculation method presented in Section 10.2.

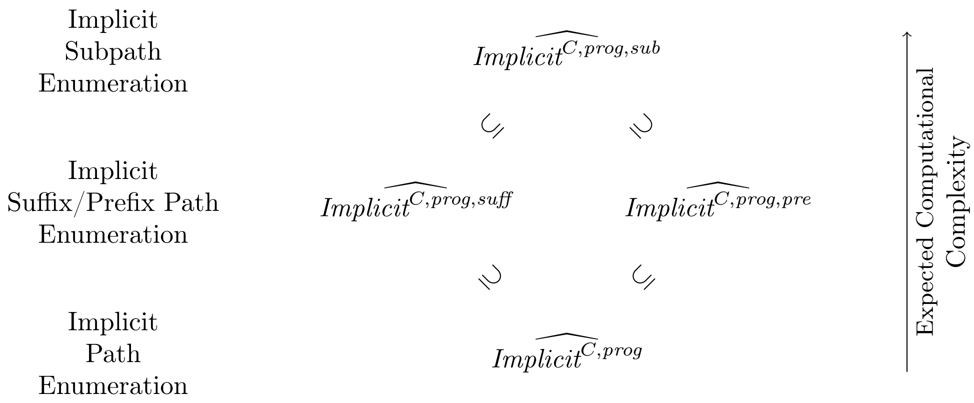

Implicit Subpath Enumeration
$\widehat{Implicit}^{C,prog,sub}$
⊆
⊇
Implicit Suffix/Prefix Path Enumeration
$\widehat{Implicit}^{C,prog,suff}$
$\widehat{Implicit}^{C,prog,pre}$
⊇
⊆
Implicit Path Enumeration
$\widehat{Implicit}^{C,prog}$
Expected Computational Complexity

Figure 10.18.: The subset relation between the sets of implicit paths considered during the different calculations of compositional base bounds provides an intuition of the expected computational complexities of the different calculations.

The first advantage is a direct consequence of the program modularity: the computationally most complex calculations of the program-modular method (i.e. implicit subpath enumeration and implicit suffix/prefix path enumeration) are performed on a per-program level. The non-modular method presented in Section 10.2, in contrast, performs an implicit enumeration of the subpaths of a graph that argues about all programs executed on the considered processor core C_j (cf. Figure 10.10). Consequently, we expect the computational complexity of the non-modular method to increase in a disproportionately high manner when increasing the number of programs executed on the considered processor core C_j. Thus, the non-modular method presented in Section 10.2 is unlikely to scale to scenarios in which multiple real-world programs are executed on the considered processor core C_j. The sketched program-modular method is more scalable as its computational complexity increases by design more or less linearly with the number of programs executed on the considered processor core C_j (i.e. the per-execution-run event bounds and compositional base bounds are calculated per program).

In addition, the sketched program-modular method is beneficial for a scenario with continuous development. In case only one of the programs executed on core C_j is changed in the course of development, the non-modular curve value calculations of Section 10.2 have to be performed from scratch. For the program-modular method, in contrast, the computationally complex calculations of per-execution-run event bounds and compositional base bounds only have to be redone for the changed programs.

Another advantage of the sketched program-modular method is that all computationally complex calculations of per-execution-run event bounds and compositional base bounds are performed statically before the timing verification (e.g. before the co-runner-sensitive WCET analysis). Thus, the complexity of these calculations is independent of the actual number of arrival curve values calculated during the timing verification. For the non-modular method presented in Section 10.2, in contrast, the computationally complex implicit subpath enumeration is performed during every calculation of an arrival curve value. Consequently, we expect that the sketched program-modular method is particularly beneficial for timing verification scenarios in which every processor core executes multiple programs and, thus, the timing verification is likely to require a relatively high number of arrival curve value calculations.

Further note that the non-modular method presented in Section 10.2 must be performed on a graph with a significantly increased edge-weight-sensitivity in order to obtain reasonably precise arrival curve values (cf. Section 6.4.4). Intuitively, if an execution of a basic block takes more clock cycles, it typically also can generate a higher amount of interference events (e.g. a certain amount of shared-bus access cycles might only be possible during the execution of a basic block in case a certain number of misses in the local cache happen and, thus, the execution of the basic block at least takes a certain amount of clock cycles). An edge-weight-insensitive graph with edge weights lower-bounding the number of clock cycles and upper-bounding the number of occurrences of interference event E (cf. Section 10.2) implicitly assumes that every execution of a considered basic block takes the minimal amount of time and, at the same time, generates the maximal amount of occurrences of interference event E. As a consequence, performing the non-modular method presented in Section 10.2 on an edge-weight-insensitive graph typically leads to a poor precision of the calculated arrival curve value. To avoid this source of imprecision, the experiments in Section 10.2, Section 10.3, and Section 10.5 rely on a graph that is fully edge-weight-sensitive in the upper bound on the number of occurrences of interference event E. The precision of the sketched program-modular method, in contrast, does not depend on the edge-weight-sensitivity of the graphs based on which the compositional base bounds are calculated as a subtraction (incorporating the upper bound on the number of occurrences of event E and the lower bound on the number of clock cycles) is already performed during the construction of the graphs (cf. Section 8.3). Thus, we expect that—even in a scenario with only a single program executed per processor core—the computational complexity of the sketched program-modular method is significantly lower than the computational complexity of the non-modular method presented in Section 10.2.

Last but not least, we would like to point out that a precise incorporation of a minimum inter-start time significantly increases the computational complexity of the non-modular method presented in Section 10.2 (cf. Section 10.3.3). For the sketched program-modular method, in contrast, a minimum inter-start time is precisely incorporated by only a single additional constraint (cf. equation (10.119)). Thus, we expect that the precise incorporation of minimum inter-start times does not significantly increase the computational complexity of the sketched program-modular method.

10.5. Evaluation in the Context of a Co-Runner-Sensitive WCET Analysis

In this section, we experimentally evaluate the presented methods for calculating arrival curve values in the context of our co-runner-sensitive WCET analysis (cf. Section 7.5). All presented methods can as well be applied in combination with interference-aware schedulability analyses. An experimental evaluation of their application in the context of interference-aware schedulability analyses, however, is beyond the scope of this thesis.

In our experiments, we calculate co-runner-sensitive WCET bounds for programs executed on a quad-core processor with a shared bus and a Round-Robin bus arbitration policy (cf. Figure 9.1). The considered processor cores shall feature an out-of-order pipeline, a local instruction cache of size 1KiB, and a local data cache of size 1KiB (i.e. processor core configuration $Conf_{ic}^{ooo}$ from Table 9.1).

In order to keep the software setup of the system under analysis simple, we assume that every processor core repeatedly executes a single program in a non-preemptive fashion. To this end, we permute the overall list of benchmarks from Table 9.2 and, subsequently, partition it into groups of four benchmarks. Table 10.10 lists the resulting twelve groups of benchmarks. Each group stands for a setting of four co-running benchmarks. Note that, in order to have exactly four benchmarks in each group, one of the benchmarks (`cjpeg_wrbmp`) has been used to fill up the last group and, thus, is a member of two groups. For this particular benchmark, we only calculate a co-runner-sensitive WCET bound for the co-runner scenario of the first group it is a member of.

Groups of Co-Running Benchmarks	
`cjpeg_wrbmp`	`sha`
`g723_enc`	`pilot`
`dijkstra`	`roboDog`
`digital_stopwatch`	`fir2dim`
`powerwindow`	`lift`
`minver`	`ndes`
`mpeg2`	`cruise_control`
`jfdctint`	`huff_dec`
`audiobeam`	`iir`
`lms`	`prime`
`trolleybus`	`adpcm_enc`
`countnegative`	`flight_control`
`h264_dec`	`petrinet`
`matrix1`	`pm`
`rijndael_enc`	`ludcmp`
`md5`	`es_lift`
`gsm_dec`	`epic`
`statemate`	`filterbank`
`gsm_encode`	`complex_updates`
`fft`	`adpcm_dec`
`binarysearch`	`cjpeg_transupp`
`susan`	`st`
`rijndael_dec`	`bsort`
`insertsort`	`cjpeg_wrbmp`

Table 10.10.: Groups of co-running benchmarks: each benchmark is executed in a scenario in which the other three benchmarks in its group are executed on the concurrent processor cores.

Note that we have not yet implemented the iterative, co-runner-sensitive WCET analysis starting from an optimistic initialization (cf. Algorithm 7.2) in our analysis prototype. Thus, we evaluate the calculation of values on arrival curves in combination with an iterative approach to co-runner-sensitive WCET analysis starting from a pessimistic initialization (cf. Algorithm 7.1). It starts from a co-runner-insensitive WCET bound and performs iterative updates of the WCET bound and the values on arrival curves until a fixed point is reached. The shared-bus interference is quantified at the granularity of granted access cycles only (cf. Section 7.6). The (re)calculations of the WCET bound are performed on graphs that are fully node-sensitive at basic block boundaries, node-insensitive inside of basic blocks, and fully edge-weight-sensitive with respect to the lower bound on the number of blocked cycles (cf. Section 6.4.4).

In this section, we conduct seven experiments. Each experiment performs a processor-core-modular, co-runner-sensitive WCET analysis (cf. Algorithm 7.1) per considered benchmark (cf. Table 10.10). The seven experiments differ in the method that is used for calculating values on arrival curves. Table 10.11 lists the seven different methods respectively variants of methods that we consider. The first experiment relies on implicit subpath enumeration with binary variables (cf. Section 10.2.3) for calculating values on arrival curves. It assumes a scheduler which does

ISPET	Implicit subpath enumeration with binary variables (cf. Section 10.2.3), no minimum inter-start times enforced.
$progGran_{0.5}$	Calculation at the granularity of program runs (cf. Section 10.3.1), assuming a relative minimum inter-start time of 0.5.
$combined_{0.5}$	Combination of a calculation at the granularity of program runs and implicit subpath enumeration with binary variables (first approach presented in Section 10.3.4), assuming a relative minimum inter-start time of 0.5.
$progGran_{0.9}$	Calculation at the granularity of program runs (cf. Section 10.3.1), assuming a relative minimum inter-start time of 0.9.
$combined_{0.9}$	Combination of a calculation at the granularity of program runs and implicit subpath enumeration with binary variables (first approach presented in Section 10.3.4), assuming a relative minimum inter-start time of 0.9.
$progGran_{0.95}$	Calculation at the granularity of program runs (cf. Section 10.3.1), assuming a relative minimum inter-start time of 0.95.
$combined_{0.95}$	Combination of a calculation at the granularity of program runs and implicit subpath enumeration with binary variables (first approach presented in Section 10.3.4), assuming a relative minimum inter-start time of 0.95.

Table 10.11.: The seven different methods respectively variants of methods that we use to calculate values on arrival curves during our co-runner-sensitive experiments.

not enforce minimum inter-start times of the programs. The second experiment performs a curve value calculation at the granularity of program runs (cf. Section 10.3.1) and assumes a relative minimum inter-start time of 0.5 (cf. equation (10.57)). The third experiment combines a curve value calculation at the granularity of program runs with implicit subpath enumeration as described at the beginning of Section 10.3.4. It also assumes a relative minimum inter-start time of 0.5. Finally, the two experiments assuming a relative minimum inter-start time of 0.5 are analogously repeated for a relative minimum inter-start time of 0.9 and a relative minimum inter-start time of 0.95 as defined by the following equations.

$$MIST_{prog_{C_j}} = WCET^{UB,0\text{-}interfer}_{prog_{C_j},C_j} + 0.9 \cdot (WCET^{UB,insens}_{prog_{C_j},C_j} - WCET^{UB,0\text{-}interfer}_{prog_{C_j},C_j}) \tag{10.174}$$

$$MIST_{prog_{C_j}} = WCET^{UB,0\text{-}interfer}_{prog_{C_j},C_j} + 0.95 \cdot (WCET^{UB,insens}_{prog_{C_j},C_j} - WCET^{UB,0\text{-}interfer}_{prog_{C_j},C_j}) \tag{10.175}$$

We conduct all experiments on a quad-core Intel® Core™ i7 processor clocked at 2.4 GHz and provided 16 GiB of main memory. The co-runner-sensitive WCET analysis is implemented by four analysis threads—one analysis thread per processor core of the system under analysis. The main analysis thread calculates WCET bounds for the considered benchmark. The three further analysis threads calculate arrival curve values for the three concurrent processor cores.

For detailed figures presenting the analysis runtime, memory consumption, and WCET bound per benchmark and experiment, we refer to Appendix B (pages 355 to 375). Note that, for the sake of comparison, the values presented there are normalized to the corresponding values of a co-runner-insensitive WCET analysis (cf. Section 7.4).

In this section, we provide tables that cumulatively summarize some of the experiment results. As a starting point, Table 10.12 shows the overall runtime of each of the seven experiments and—for the sake of comparison—also the overall runtime of the corresponding co-runner-insensitive experiment (cf. Figure B.5). The table demonstrates that each co-runner-sensitive experiment takes significantly longer than the co-runner-insensitive experiment (at least more than four times as long). This is expected as the co-runner-sensitive experiments consider the operation of all four

	Overall Experiment Runtime			
co-runner-insensitive		1h	16m	5.09s
ISPET		12h	16m	25.35s
$\mathit{progGran}_{0.5}$		5h	42m	20.6s
$\mathit{combined}_{0.5}$		16h	10m	58.55s
$\mathit{progGran}_{0.9}$		6h	17m	49.78s
$\mathit{combined}_{0.9}$	1d	3h	45m	2.02s
$\mathit{progGran}_{0.95}$		7h	23m	9.9s
$\mathit{combined}_{0.95}$	1d	12h	42m	22.2s

Table 10.12.: Overall runtime of the conducted co-runner-sensitive experiments and the corresponding co-runner-insensitive experiment.

processor cores while the co-runner-insensitive experiment only considers the operation of a single processor core. In addition, the table shows that the co-runner-sensitive experiments relying on implicit subpath enumeration for calculating values on arrival curves (*ISPET*, $\mathit{combined}_{0.5}$, $\mathit{combined}_{0.9}$, and $\mathit{combined}_{0.95}$) take significantly longer than the co-runner-sensitive experiments which only calculate arrival curve values at the granularity of program runs. This is also expected as the computational complexity of calculating arrival curve values via implicit subpath enumeration is significantly higher than the computational complexity of a curve value calculation at the granularity of program runs (cf. Section 10.1 to Section 10.3). Finally, Table 10.12 demonstrates that increasing the relative minimum inter-start time also increases the experiment runtime. This is expected, too, as increasing the relative (and, thus, also the absolute) minimum inter-start time potentially results in co-runner-sensitive WCET bounds that are smaller. Consequently, the iterative analysis approach (starting from a co-runner-insensitive WCET bound) typically needs a higher number of iterations until a fixed point is reached.

Next, we investigate whether the co-runner-sensitive experiments are able to decrease the WCET bounds compared to the co-runner-insensitive experiment. There are twelve out of 47 benchmarks for which one of the co-runner-sensitive experiments calculates a WCET bound that is smaller than the corresponding co-runner-insensitive WCET bound. Table 10.13 lists the WCET bounds calculated by the seven co-runner-sensitive experiments for each of these twelve benchmarks. Note that each presented WCET bound is normalized to the corresponding co-runner-insensitive WCET bound.

Table 10.13 shows that the co-runner-sensitive experiments assuming a relative minimum inter-start time of at least 0.9 are able to improve the WCET bounds of some benchmarks (i.e. they provide a normalized value smaller than one for these benchmarks). For the experiments assuming a relative minimum inter-start time of 0.9, there are eight out of 47 benchmarks (i.e. 17 percent of the benchmarks) for which the calculated co-runner-sensitive WCET bound is (by up to 16.2 percent) smaller than the corresponding co-runner-insensitive WCET bound. For the experiments assuming a relative minimum inter-start time of 0.95, there are twelve out of 47 benchmarks (i.e. 25.5 percent of the benchmarks) for which the calculated co-runner-sensitive WCET bound is (by up to 22 percent) smaller than the corresponding co-runner-insensitive WCET bound.

These results are a bit disappointing as we need to enforce relatively high minimum inter-start times (i.e. the scheduler makes sure that every program execution run is delayed up to 95 percent of the interference that is considered by the co-runner-insensitive analysis) in order to reduce the WCET bound compared to a co-runner-insensitive WCET bound for only around a quarter of the benchmarks. Note, however, that the setup considered during the co-runner-sensitive experiments is simplified in the sense that every processor core repeatedly executes a single program over and over again. In this context, the enforcement of a minimum inter-start time can be seen as a very simple way to adjust the utilization of a processor core (and, thus, also the amount of

	ISPET	*progGran*$_{0.5}$	*combined*$_{0.5}$	*progGran*$_{0.9}$	*combined*$_{0.9}$	*progGran*$_{0.95}$	*combined*$_{0.95}$
`flight_control`	1.000	1.000	1.000	0.838	0.838	0.780	0.780
`powerwindow`	1.000	1.000	1.000	0.961	0.961	0.929	0.929
`fft`	1.000	1.000	1.000	0.999	0.999	0.971	0.960
`md5`	1.000	1.000	1.000	0.932	0.932	0.907	0.907
`sha`	1.000	1.000	1.000	1.000	1.000	0.979	0.979
`cjpeg_transupp`	1.000	1.000	1.000	0.884	0.884	0.823	0.821
`dijkstra`	1.000	1.000	1.000	0.936	0.936	0.865	0.865
`g723_enc`	1.000	1.000	1.000	1.000	1.000	0.995	0.995
`mpeg2`	1.000	1.000	1.000	1.000	1.000	0.998	0.998
`rijndael_dec`	1.000	1.000	1.000	0.995	0.995	0.961	0.961
`rijndael_enc`	1.000	1.000	1.000	0.938	0.938	0.910	0.910
`susan`	1.000	1.000	1.000	1.000	1.000	0.975	0.975

Table 10.13.: WCET bounds calculated by the co-runner-sensitive experiments normalized to the corresponding co-runner-insensitive WCET bounds. There are twelve out of 47 benchmarks for which a co-runner-sensitive experiment calculates a WCET bound that is smaller than the corresponding co-runner-insensitive WCET bound.

shared-resource interference that the core contributes). Thus, for more significant results, we recommend to experimentally evaluate the different methods for calculating arrival curve values in the context of a real-world interference-aware schedulability analysis. Such an experimental evaluation, however, is beyond the scope of this thesis.

A closer consideration might also lead to the insight that all processor-core-modular co-runner-sensitive approaches to timing verification are by design hardly more precise than co-runner-insensitive approaches to timing verification. Recall that processor-core-modular approaches inherently sacrifice a certain amount of precision in order to avoid the enumeration of all possible interleavings of shared-resource access requests by the different processor cores (cf. Section 2.2.2). The principle behind processor-core-modular approaches to timing verification is very similar to the principle behind thread-modular schemes used in the verification of multi-threaded software [Flanagan and Qadeer, 2003; Henzinger et al., 2003; Malkis et al., 2007; Gotsman et al., 2007; Miné, 2011, 2014; Monat and Miné, 2017]. It is, however, known that thread-modular verification techniques only work well for multi-threaded software in which the operation of the threads is *loosely coupled* (i.e. they only make very limited use of shared variables for inter-thread communication) [Flanagan and Qadeer, 2003]. The operation of the different processor cores in a multi-core processor, in contrast, is *closely coupled* by the resource sharing. Thus, it would not be surprising if the inherent imprecision of processor-core-modular co-runner-sensitive approaches to timing verification was so high that these approaches are hardly more precise than co-runner-insensitive approaches. To the best of our knowledge, it remains an open question how precise processor-core-modular co-runner-sensitive approaches to timing verification can at best be (i.e. which degree of precision loss is inherent to the processor-core-modular scheme).

We would also like to point out that, so far, there is a clear point in using processor-core-modular co-runner-sensitive approaches to timing verification: there are scenarios in which a co-runner-insensitive timing verification is not possible. This is e.g. the case for systems with multi-core processors, shared buses, and priority-based bus arbitration due to the possibility of *starvation* (cf. Section 7.7). In such scenarios, a co-runner-sensitive approach is mandatory. To the best of our knowledge, so far, processor-core-modular approaches are the only co-runner-sensitive approaches to timing verification known to scale to real-world software and hardware platforms (cf. Section 2.2).

Coming back to the results presented in Table 10.13, we also see that the experiments making use of implicit subpath enumeration for the calculation of arrival curve values ($\mathit{combined}_{0.5}$, $\mathit{combined}_{0.9}$, $\mathit{combined}_{0.95}$) hardly result in more precise WCET bounds than the corresponding experiments only resorting to a calculation at the granularity of program runs ($\mathit{progGran}_{0.5}$, $\mathit{progGran}_{0.9}$, $\mathit{progGran}_{0.95}$). For the experiments assuming a relative minimum inter-start time of 0.5 or 0.9, there is no difference at all between the co-runner-sensitive WCET bounds calculated with and without implicit subpath enumeration. For the experiments assuming a relative minimum inter-start time of 0.95, there are only two benchmarks for which the incorporation of implicit subpath enumeration slightly decreases the co-runner-sensitive WCET bounds by up to 1.13 percent (for benchmark `fft`: $0.960/0.971 = 0.9887 = 1 - 0.0113$).

In Section 10.3.4, we observe an average reduction of 5.2 percent for the arrival curve values calculated by additionally incorporating implicit subpath enumeration. This improvement of precision during the calculation of arrival curve values, however, does not seem to lead to a significantly improved precision of the co-runner-sensitive WCET analysis. Note, however, that the results reported in Section 10.3.4 assume a relative minimum inter-start time of 0.5. In addition, we would like to point out that it heavily depends on the considered benchmark and the considered interval lengths how much precision can be gained during the calculation of arrival curve values by additionally incorporating implicit subpath enumeration. For more significant and reliable results on the potential precision gain by additionally incorporating implicit subpath enumeration during co-runner-sensitive timing verification, we recommend to experimentally evaluate the different methods for calculating arrival curve values in the context of a real-world interference-aware schedulability analysis. As mentioned before, such an experimental evaluation is beyond the scope of this thesis.

Part IV.

Closure

Chapter 11 Conclusion

The end is near, the end is coming.

(Day of the Rapture, Plague Cycle, 2017)

The main technical contributions of this thesis are listed in Chapter 3. In the current chapter, in contrast, we point out in which sense the overall formal framework described throughout this thesis helps to complete the big picture in the area of research on WCET analysis. Moreover, we discuss what the experimental results presented in this thesis mean for the future of research on WCET analysis for systems with multi-core processors.

11.1. Relevance of Our Formal Framework

The *property lifting framework* (cf. Chapter 4) can be seen as a *toolbox* for the derivation of analysis and verification approaches. In Chapter 5 and Chapter 6, we demonstrate the application of property lifting during the formal derivation of the well-known calculation of WCET bounds based on abstract interpretation [Cousot and Cousot, 1977] and implicit path enumeration [Li and Malik, 1995; Puschner and Schedl, 1997; Stein, 2010]. The soundness of the underlying abstract interpretation [Thesing, 2004] is assumed as a starting point for the formal derivation. Thus, the overall soundness argument is based on abstract interpretation and property lifting. In this context, the property lifting framework can be seen as complementing the abstract interpretation framework rather than replacing it.

Chapter 5 and Chapter 6 also show that it is very time-consuming and tedious to instantiate the property lifting framework for the three levels of approximation used during the calculation of WCET bounds. We argue, however, that the result of this instantiation can be seen as an *extended toolbox* which is already prepared for the specific needs during the derivation of WCET analyses and similar analyses calculating event bounds. Hence, we expect that this extended toolbox can mostly be reused for the derivation of future WCET analyses.

In Chapter 7, we demonstrate how to use this extended toolbox to formally derive WCET analyses for multi-core processors with shared buses. The derivation of the co-runner-insensitive WCET analysis features a *full soundness proof.* For some of the presented co-runner-sensitive WCET analyses, however, we omit the soundness proofs due to time and space constraints. In Chapter 8, we use the extended toolbox to specify approaches to the calculation of compositional base bounds. We conclude that the application of our extended toolbox can also be of use in scenarios in which there is no need for a full soundness proof: The extended toolbox *structures the derivation* of analyses by its unified lifting workflow (even if the lifting steps are not proven). In addition, the underlying algebra provides a *clean and concise specification* of the assumed system properties and the resulting bound calculations.

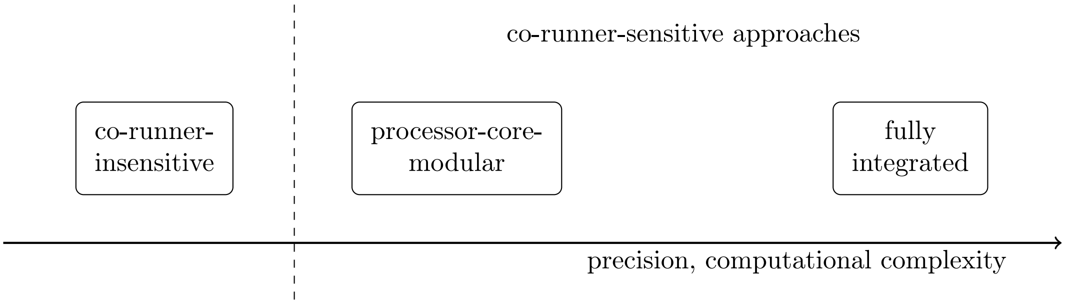

co-runner-sensitive approaches
co-runner-insensitive
processor-core-modular
fully integrated
precision, computational complexity

Figure 11.1.: Precision and Computational Complexity of the Three Known Classes of Approaches to Timing Verification for Systems with Multi-Core Processors and Unpartitioned Shared Resources

11.2. Computational Complexity of WCET Analysis for Multi-Core Processors

In Chapter 2, we survey a wide range of approaches to timing verification for systems with multi-core processors. For systems with unpartitioned shared resources, we identify three classes of existing approaches (cf. Section 2.2): co-runner-insensitive (i.e. Murphy) approaches, processor-core-modular approaches, and fully integrated approaches. The processor-core-modular and fully integrated approaches are so far the only known co-runner-sensitive approaches. The three classes of approaches differ in precision and computational complexity—with a higher precision typically implying a higher computational complexity (cf. Figure 11.1).

We would like to point out that the formal derivation of WCET analyses for multi-core processors presented in Chapter 7 also quite *naturally explains the expected computational complexity*. The *co-runner-insensitive* analysis (cf. Section 7.4) is based on a single abstract model that only considers the operation of one processor core in detail and does not make any assumptions about the programs executed on the concurrent cores. Thus, it has a relatively low computational complexity. The *processor-core-modular* analyses, in contrast, feature one such abstract model per processor core (cf. Section 7.5). The iterative processor-core-modular algorithms only exchange cumulative information (in our case WCET bounds and arrival curve values) between the abstract models of the different processor cores (cf. Algorithm 7.1 and Algorithm 7.2). Our experiments in Section 10.5 show that the co-runner-sensitive processor-core-modular experiments take between four and 29 times as long as the corresponding co-runner-insensitive experiment for a quad-core processor (depending on the configured minimum inter-start time and the complexity of the applied calculation of arrival curve values, cf. Table 10.12). Intuitively, the complexity of the micro-architectural analysis adds up for the different processor cores. Thus, the computational complexity of the micro-architectural analysis during a processor-core-modular consideration is expected to scale more or less linearly in the number of processor cores. The results of our experiments, however, only show a very limited gain in precision for the processor-core-modular analyses compared to the co-runner-insensitive analysis (cf. Table 10.13). Finally, a *fully integrated* analysis operates on the cross product of the abstract models of all processor cores (cf. equation (7.20)) modulo pruned compound abstract traces (cf. equation (7.26)). Thus, the computational complexity of fully integrated analyses is expected to scale close to exponentially in the number of processor cores. We have not implemented or evaluated a fully integrated analysis, but existing experiment results [Kelter, 2015] and the lack of such results [Giannopoulou et al., 2012; Lampka et al., 2014] strongly indicate a poor scalability (cf. Section 2.2.1). In a nutshell, we think that the formal derivation of WCET analyses for multi-core processors presented in Chapter 7 makes an important contribution to understanding the inherent computational complexity of different approaches to WCET analysis for multi-core processors.

11.3. Results of the Processor-Core-Modular Experiments

The precision of the experimental results presented in Section 10.5 is disappointing: In our experiments, the co-runner-sensitive processor-core-modular analysis is hardly more precise than the co-runner-insensitive analysis. There are scenarios of certain co-running benchmarks for which the processor-core-modular analysis is more precise than the co-runner-insensitive analysis [Jacobs et al., 2015], for most scenarios of co-running benchmarks in our experiments, however, this does not seem to be the case.

These disappointing experimental results raise the question how it is possible that we are the first researchers to report that processor-core-modular timing verification barely improves the precision compared to co-runner-insensitive timing verification. Processor-core-modular timing verification has been the de facto standard for scalable co-runner-sensitive timing verification for roughly a decade (cf. Section 2.2.2)—although the common level of modularity of all these approaches has not been recognized until now. To the best of our knowledge, all existing processor-core-modular approaches rely on cumulative approximations at best as precise as ours. This leads to a number of follow-up questions:

- Have the proposed processor-core-modular approaches been compared to co-runner-insensitive approaches with respect to analysis precision?

- Were the proposed processor-core-modular approaches experimentally evaluated for an over-provisioned hardware platform?

- Were there other irregularities in the considered experimental setups that effectively made the comparison to the results of a co-runner-insensitive approach unfair?

Answering some of these questions, however, is particularly challenging as the details of the considered experiment setups are typically not available and researchers are usually not very cooperative when it comes to putting in doubt their published research results. On the other hand, it is as well possible that there is a conceptual problem in our evaluation methodology.

Note that we are not yet fully confident in the universality of the presented experiment results: Due to time and space constraints, we only evaluated a more or less synthetic scenario in which every processor core repeatedly executes a single program. Consequently, future experiments would have to consider more complex scenarios of processor-core-modular interference-aware schedulability analyses. Moreover, we only evaluated scenarios with a multi-core processor with a shared bus. Thus, it is not clear whether the additional consideration of (potentially multi-level) shared caches would lead to similarly disappointing results with respect to analysis precision. Additionally, we would like to point out that the cumulative approximation used in our processor-core-modular analysis (WCET bounds and arrival curve values) is fairly simple. It remains an open question whether there are sophisticated cumulative approximations leading to more precise analysis results.

However, be aware that there are hardware platforms for which a co-runner-insensitive timing verification is not possible (cf. Section 7.7). For such hardware platforms, a co-runner-sensitive timing verification is mandatory and processor-core-modular approaches are currently the only known co-runner-sensitive approaches for which a scaling to real-world scenarios is not completely out of reach.

11.4. Future of WCET Analysis for Multi-Core Processors

Co-runner-insensitive approaches to WCET analysis for multi-core processors are inherently pessimistic as they do not take into account the operation of the programs executed on the concurrent processor cores. As a consequence, their use will likely have to be compensated by a significant over-provisioning of the hardware platform executing the application under analysis. Such an over-provisioning increases the price, the weight, and the energy consumption of the hardware platform. Hence, such an over-provisioning is unsustainable and might easily outweigh the advantages of using a multi-core platform compared to using multiple dedicated single-core systems.

The other extreme approach to WCET analysis for multi-core processors with unpartitioned shared resources is a fully integrated approach (cf. Figure 11.1), which is very unlikely to scale to real-world scenarios (cf. Section 2.2.1). Thus, it is not surprising that many researchers rely on processor-core-modular timing verification as a compromise between aforementioned extreme approaches (cf. Section 2.2.2).

The experimental results presented in Section 10.5, however, indicate that there are at least certain execution scenarios in which processor-core-modular approaches are almost equally pessimistic as co-runner-insensitive approaches. In our opinion, a significant amount of the pessimism in processor-core-modular approaches is due to the loose coupling of the abstract models of the different processor cores, which only exchange accumulated information (in our case WCET bounds and arrival curve values). On the other hand, this loose coupling is what makes processor-core-modular approaches significantly more scalable than fully integrated approaches. Thus, we believe that a significant amount of the pessimism in processor-core-modular approaches is inherent to the processor-core-modularity.

Based on the presented experimental results, we do not believe that processor-core-modular timing verification alone can lead to significantly more precise results than co-runner-insensitive timing verification. Instead, for systems with unpartitioned shared resources, we recommend to combine processor-core-modular approaches with *traffic shaping* of the accesses to the shared resources [Georgiadis et al., 1996; Davis and Navet, 2012; Oehlert et al., 2019]. In this way, the pessimism of the processor-core-modularity might be alleviated without having to resort to static resource schedules, which potentially require expensive offline optimizations for good results [Rosen et al., 2007; John and Jacobs, 2014]. In this context, traffic shaping (i.e. making sure at runtime that a certain arrival curve is never exceeded by the shared-resource access behavior of a processor core) can be seen as a more dynamic and indirect way of partitioning the shared resources.

11.5. Related Work in Our Group

This thesis and the thesis of Sebastian Hahn [Hahn, 2018] have been created in close cooperation. The contributions of both theses can be seen as complementary in the sense that they focus on different aspects with respect to the state of the art in WCET analysis. This thesis focuses on the principle of property lifting and its instantiation for WCET analysis—in general as well as for multi-core processors. The thesis of Sebastian Hahn, in contrast, focuses on timing compositionality and analysis approaches and hardware designs particularly appropriate for a timing-compositional consideration. Both theses assume—as most work in the area of static WCET analysis—that the full specification of the considered micro-architecture is known. We are aware of the fact that this assumption is currently not realistic for commercial hardware platforms. We are, however, confident that a derivation of reliable verification techniques for safety- and/or mission-critical application scenarios is not possible without this assumption. In order to partially close this gap, our group tries to automatically and safely determine certain hardware parameters (as e.g. the size and latency of caches) for a given micro-architecture based on an incomplete hardware specification [Abel and Reineke, 2013, 2019].

But time flows like a river. . .
and history repeats. . .

(Secret of Mana, Square, 1993)

Part V.

Appendices

Appendix A

Additional Proofs

So zerbricht man sich den Kopf
Und es kommt oft vor dass es heißt
Für alle die es wissen wollen
Hier ist der Beweis

(Hier ist der Beweis, Tocotronic, 2002)

This appendix chapter contains the lengthy proofs of this thesis.

Proof of Statement (4.24).

$$
\begin{aligned}
& \pi^{M_a}(\widehat{LessTraces}) \\
\underset{\substack{(4.13)\\(4.23)}}{\supseteq}\; & \pi^{M_a}(\widehat{Traces} \setminus \widehat{Infeas}) \\
\underset{(4.4)}{=}\; & \pi^{M_a}(\,\{\widehat{t} \in \widehat{Traces} \mid \gamma_{trace}(\widehat{t}) \cap Traces \neq \emptyset\}\,) \\
\underset{(4.20)}{=}\; & \{\pi^{M_a}_{trace}(\widehat{t}) \mid \widehat{t} \in \widehat{Traces} \wedge \gamma_{trace}(\widehat{t}) \cap Traces \neq \emptyset\} \\
\underset{\substack{(4.17)\\(4.18)}}{=}\; & \{\widehat{t^{M_a}} \mid (\widehat{t^{M_1}}, \ldots, \widehat{t^{M_a}}, \ldots, \widehat{t^{M_m}}) \in \widehat{Traces}^{M_1} \times \ldots \times \widehat{Traces}^{M_m} \\
& \qquad \wedge \gamma_{trace}(\widehat{t^{M_1}}, \ldots, \widehat{t^{M_a}}, \ldots, \widehat{t^{M_m}}) \cap Traces \neq \emptyset\} \\
\underset{\substack{(4.19)\\(4.18)}}{=}\; & \{\widehat{t^{M_a}} \mid (\widehat{t^{M_1}}, \ldots, \widehat{t^{M_a}}, \ldots, \widehat{t^{M_m}}) \in \widehat{Traces}^{M_1} \times \ldots \times \widehat{Traces}^{M_m} \\
& \qquad \wedge \bigcap\nolimits_{M_b \in Models} \gamma^{M_b}_{trace}(\widehat{t^{M_b}}) \cap Traces \neq \emptyset\} \\
\underset{(4.3)}{=}\; & \{\widehat{t^{M_a}} \in \widehat{Traces}^{M_a} \mid \gamma^{M_a}_{trace}(\widehat{t^{M_a}}) \cap Traces \neq \emptyset\} \\
\underset{(4.4)}{=}\; & \widehat{Traces}^{M_a} \setminus \widehat{Infeas}^{M_a} \quad \square
\end{aligned}
$$

Proof of Hypothesis (4.H1). The following auxiliary statement is a direct consequence of equations (4.28) and (4.30).

$$\forall M_a \in Models : \widehat{Approx}^{M_a} \subseteq \widehat{Traces}^{M_a} \tag{A.1}$$

As a consequence of (4.21), the claim in (4.H1) trivially holds for the initial values of the approximation variables as specified in (4.28). For the general case, however, we assume the hypothesis (4.H1) to hold after a given sequence of approximation variable updates. In an inductive way, we can use this assumption to show that the hypothesis is preserved by an additional simultaneous update of an arbitrarily chosen set of the approximation variables. For the details of this induction step, please refer to (A.2) and (A.3). The inequation chain (A.2) shows that all sets contained in it are in fact equal.

$$
\begin{aligned}
&\pi^{M_a}(\widehat{\mathit{LessTraces}})\\
\underset{(4.20)}{=}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{LessTraces}}\}\\
\underset{(4.12)}{=}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{LessTraces}} \wedge \widehat{P}(\widehat{t})\}\\
\underset{(4.22)}{\subseteq}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \pi^{M_1}(\widehat{\mathit{LessTraces}}) \times \ldots \times \pi^{M_m}(\widehat{\mathit{LessTraces}}) \wedge \widehat{P}(\widehat{t})\}\\
\underset{(4.\mathrm{H}1)}{\subseteq}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{Approx}}^{M_1} \times \ldots \times \widehat{\mathit{Approx}}^{M_m} \wedge \widehat{P}(\widehat{t})\}\\
\underset{(\mathrm{A}.1)}{\subseteq}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{Traces}}^{M_1} \times \ldots \times \widehat{\mathit{Traces}}^{M_m} \wedge \widehat{P}(\widehat{t})\}\\
\underset{(4.17)}{=}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{Traces}} \wedge \widehat{P}(\widehat{t})\}\\
\underset{(4.12)}{=}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{LessTraces}}\}
\end{aligned}
\tag{A.2}
$$

This information is used in (A.3) to show that $F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}})$ is guaranteed to be a superset of the projections $\pi^{M_a}(\widehat{\mathit{LessTraces}})$.

$$
\begin{aligned}
&\pi^{M_a}(\widehat{\mathit{LessTraces}})\\
\underset{(\mathrm{A}.2)}{=}\;&\{\pi^{M_a}_{\mathit{trace}}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{Approx}}^{M_1} \times \ldots \times \widehat{\mathit{Approx}}^{M_m} \wedge \widehat{P}(\widehat{t})\}\\
\underset{(4.18)}{=}\;&\{\widehat{t^{M_a}} \mid (\widehat{t^{M_1}}, \ldots, \widehat{t^{M_a}}, \ldots, \widehat{t^{M_m}}) \in \widehat{\mathit{Approx}}^{M_1} \times \ldots \times \widehat{\mathit{Approx}}^{M_m}\\
&\qquad \wedge \widehat{P}(\widehat{t^{M_1}}, \ldots, \widehat{t^{M_a}}, \ldots, \widehat{t^{M_m}})\}\\
\underset{(4.\mathrm{C}2)}{\subseteq}\;&\{\widehat{t^{M_a}} \in \widehat{\mathit{Approx}}^{M_a} \mid \widetilde{P^{M_a}}(\widehat{t^{M_a}}, \overrightarrow{\mathit{Approx}_{\overline{M_a}}})\}\\
\underset{(\mathrm{A}.1)}{\subseteq}\;&\{\widehat{t^{M_a}} \in \widehat{\mathit{Traces}}^{M_a} \mid \widetilde{P^{M_a}}(\widehat{t^{M_a}}, \overrightarrow{\mathit{Approx}_{\overline{M_a}}})\}\\
\underset{(4.30)}{=}\;&F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}})
\end{aligned}
\tag{A.3}
$$

According to the equation system given by (4.29), each approximation variable $\widehat{\mathit{Approx}}^{M_a}$ is updated to the value $F^{M_a}(F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}}))$. This proves that the simultaneous update of an arbitrarily chosen set of approximation variables is guaranteed to preserve the hypothesis given by (4.H1). □

Proof of Hypothesis (4.H2). According to (4.28) and (4.29), hypothesis (4.H2) trivially holds for the approximation variables directly after their initialization.

For the inductive step, assume that hypothesis (4.H2) holds for a given vector of approximation variables $\widehat{Approx}^{M_a}$. Let the updated value of an approximation variable $\widehat{Approx}^{M_a}$ after the simultaneous update of an arbitrarily chosen set of approximation variables be denoted by $\widehat{Approx'}^{M_a}$. Thus, the following statement holds.

$$\forall M_a \in Models : \widehat{Approx'}^{M_a} \in \{\widehat{Approx}^{M_a}, F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}})\} \tag{A.4}$$

It follows from (4.H2) and (A.4) that each $\widehat{Approx'}^{M_a}$ is a subset of the corresponding $\widehat{Approx}^{M_a}$.

$$\forall M_a \in Models : \widehat{Approx'}^{M_a} \subseteq \widehat{Approx}^{M_a} \tag{A.5}$$

Equation (A.6) shows that $F^{M_a}(\overrightarrow{Approx'_{\overline{M_a}}})$ is a subset of $F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}})$.

$$\begin{aligned}
& F^{M_a}(\overrightarrow{Approx'_{\overline{M_a}}}) \\
\underset{(4.30)}{=} \quad & \{\widehat{t^{M_a}} \in \widehat{Traces}^{M_a} \mid \widetilde{P}^{M_a}(\widehat{t^{M_a}}, \overrightarrow{Approx'_{\overline{M_a}}})\} \\
\underset{\substack{(4.C3)\\(A.5)}}{\subseteq} \quad & \{\widehat{t^{M_a}} \in \widehat{Traces}^{M_a} \mid \widetilde{P}^{M_a}(\widehat{t^{M_a}}, \overrightarrow{Approx_{\overline{M_a}}})\} \\
\underset{(4.30)}{=} \quad & F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}})
\end{aligned} \tag{A.6}$$

Based on these results, it is straightforward to show that hypothesis (4.H2) also holds for the updated value $\widehat{Approx'}^{M_a}$ of an approximation variable, which concludes the inductive proof.

$$\begin{aligned}
& \widehat{Approx'}^{M_a} \\
\underset{(A.4)}{=} \quad & \begin{cases} \widehat{Approx}^{M_a} \\ F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) \end{cases} \\
\underset{(4.H2)}{\supseteq} \quad & F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) \\
\underset{(A.6)}{\supseteq} \quad & F^{M_a}(\overrightarrow{Approx'_{\overline{M_a}}}) \quad \square
\end{aligned} \tag{A.7}$$

Proof that any fixed point reached by our iteration starting from a maximally pessimistic initialization is guaranteed to coincide with the (unique) greatest fixed point of equation system (4.33). Note that, during this proof, we rely on the syntactical convention that for any vector $\overrightarrow{Some}$ with a dedicated position per abstract model M_a there are corresponding vectors $\overrightarrow{Some_{\overline{M_a}}}$ in which the element for M_a has been removed.

$$\begin{aligned}
& \overrightarrow{Some} = (\widehat{Some}^{M_1}, \ldots, \widehat{Some}^{M_m}) \\
\Rightarrow & \forall M_a \in Models : \overrightarrow{Some_{\overline{M_a}}} = (\widehat{Some}^{M_1}, \ldots, \widehat{Some}^{M_{a-1}}, \widehat{Some}^{M_{a+1}}, \ldots, \widehat{Some}^{M_m})
\end{aligned} \tag{A.8}$$

By design (cf. equations (4.28), (4.29), and (4.30)), each approximation variable can be seen as a member of a power set domain.

$$\forall M_a \in Models : \widehat{Approx}^{M_a} \in \mathcal{P}(\widehat{Traces}^{M_a}) \tag{A.9}$$

Moreover, we know that each power set domain forms a complete lattice in combination with the relation $\subseteq$.

$$\forall M_a \in \mathit{Models} : (\mathcal{P}(\widehat{\mathit{Traces}}^{M_a}), \subseteq) \text{ is a complete lattice} \tag{A.10}$$

Thus, the cross product over these power set domains also forms a complete lattice in combination with the pairwise extension of $\subseteq$ (cf. e.g. Appendix A.2 of [Nielson et al., 1999]). In the following, we use D_1 to refer to this cross product.

$$D_1 := \mathcal{P}(\widehat{\mathit{Traces}}^{M_1}) \times \ldots \times \mathcal{P}(\widehat{\mathit{Traces}}^{M_m}) \tag{A.11}$$

$$(D_1, \subseteq_{\mathit{pairwise}}) \text{ is a complete lattice} \tag{A.12}$$

The simultaneous update of all approximation variables can be defined as a function F on D_1. Note that—by design—the set of fixed points of equation system (4.33) coincides with the set of all fixed points of function F.

$$F : D_1 \to D_1 \tag{A.13}$$

$$F(\overrightarrow{\mathit{Approx}}) = (F^{M_1}(\overrightarrow{\mathit{Approx}_{\overline{M_1}}}), \ldots, F^{M_m}(\overrightarrow{\mathit{Approx}_{\overline{M_m}}})) \tag{A.14}$$

The monotonicity of the update functions F^{M_a} (cf. criterion (4.C3)) implies the monotonicity of function F.

$$\forall \overrightarrow{\mathit{Approx}}, \overrightarrow{\mathit{Approx}'} \in D_1 : \overrightarrow{\mathit{Approx}'} \subseteq_{\mathit{pairwise}} \overrightarrow{\mathit{Approx}} \Rightarrow F(\overrightarrow{\mathit{Approx}'}) \subseteq_{\mathit{pairwise}} F(\overrightarrow{\mathit{Approx}}) \tag{A.15}$$

It follows from this monotonicity and $(D_1, \subseteq_{\mathit{pairwise}})$ being a complete lattice that F (respectively equation system (4.33)) has a unique greatest fixed point [Tarski, 1955]. This greatest fixed point shall be referred to as $\overrightarrow{\mathit{GFP}}$.

$$\overrightarrow{\mathit{GFP}} = (\widehat{\mathit{GFP}}^{M_1}, \ldots, \widehat{\mathit{GFP}}^{M_m}) \tag{A.16}$$

Note that the maximally pessimistic initialization of our iterative approach is a member of D_1 and by construction greater than or equal to the greatest fixed point.

$$\overrightarrow{\mathit{GFP}} \subseteq_{\mathit{pairwise}} (\widehat{\mathit{Traces}}^{M_1}, \ldots, \widehat{\mathit{Traces}}^{M_m}) \tag{A.17}$$

Moreover, we know due to the monotonicity of the update functions F^{M_a} that whenever a vector of approximation variables is greater than or equal to the greatest fixed point, the update of an approximation variable cannot let its value jump below the corresponding component of the greatest fixed point.

$$\begin{aligned} &\overrightarrow{\mathit{GFP}} \subseteq_{\mathit{pairwise}} \overrightarrow{\mathit{Approx}} \\ \Rightarrow &\forall M_a \in \mathit{Models} : \widehat{\mathit{GFP}}^{M_a} = F^{M_a}(\overrightarrow{\mathit{GFP}_{\overline{M_a}}}) \subseteq F^{M_a}(\overrightarrow{\mathit{Approx}_{\overline{M_a}}}) \end{aligned} \tag{A.18}$$

As a consequence, starting from a maximally pessimistic initialization, after any sequence of updates of the approximation variables every variable is guaranteed to be a superset of or equal to the corresponding component of the greatest fixed point. This implies that any fixed point reached by our iterative approach starting from a maximally pessimistic initialization has to coincide with the greatest fixed point of equation system (4.33). □

Proof that any fixed point reached by the approach in Section 4.2.4 is guaranteed to coincide with the (unique) least fixed point of equation system (4.37) *and that no sequence of updates applied to the initialization can exceed this fixed point.* Note that, during this proof, we rely on the syntactical convention that for any vector $\overrightarrow{Some}$ with a dedicated position per abstract model M_a there are corresponding vectors $\overrightarrow{Some_{\overline{M_a}}}$ in which the element for M_a has been removed.

$$\begin{aligned}&\overrightarrow{Some} = (\widehat{Some}^{M_1}, \dots, \widehat{Some}^{M_m})\\ \Rightarrow &\forall M_a \in \mathit{Models} : \overrightarrow{Some_{\overline{M_a}}} = (\widehat{Some}^{M_1}, \dots, \widehat{Some}^{M_{a-1}}, \widehat{Some}^{M_{a+1}}, \dots, \widehat{Some}^{M_m})\end{aligned} \tag{A.19}$$

By design (cf. equations (4.36), (4.29), (4.30), and (4.H3)), each approximation variable $\widehat{Approx}^{M_a}$ can be seen as a member of domain $\widehat{Dom}^{M_a}$. $\widehat{Dom}^{M_a}$ contains the members of $\mathcal{P}(\widehat{Traces}^{M_a})$ that are a superset of or equal to $\widehat{Init}^{M_a}$.

$$\widehat{Dom}^{M_a} = \{set \mid \widehat{Init}^{M_a} \subseteq set \subseteq \widehat{Traces}^{M_a}\} \tag{A.20}$$

$$\forall M_a \in \mathit{Models} : \widehat{Approx}^{M_a} \in \widehat{Dom}^{M_a} \tag{A.21}$$

In the same way as $\mathcal{P}(\widehat{Traces}^{M_a})$, the domain $\widehat{Dom}^{M_a}$ forms a complete lattice in combination with the relation $\subseteq$.

$$\forall M_a \in \mathit{Models} : (\widehat{Dom}^{M_a}, \subseteq) \text{ is a complete lattice} \tag{A.22}$$

Thus, the cross product over the domains $\widehat{Dom}^{M_a}$ also forms a complete lattice in combination with the pairwise extension of $\subseteq$ (cf. e.g. Appendix A.2 of [Nielson et al., 1999]). In the following, we use D_2 to refer to this cross product.

$$D_2 := \widehat{Dom}^{M_1} \times \dots \times \widehat{Dom}^{M_m} \tag{A.23}$$

$$(D_2, \subseteq_{pairwise}) \text{ is a complete lattice} \tag{A.24}$$

The simultaneous update of all approximation variables can be defined as a function F on D_2.

$$F : D_2 \to D_2 \tag{A.25}$$

$$F(\overrightarrow{Approx}) = (F^{M_1}(\overrightarrow{Approx_{\overline{M_1}}}), \dots, F^{M_m}(\overrightarrow{Approx_{\overline{M_m}}})) \tag{A.26}$$

The monotonicity of the update functions F^{M_a} (cf. criterion (4.C3)) implies the monotonicity of function F.

$$\forall \overrightarrow{Approx}, \overrightarrow{Approx'} \in D_2 : \overrightarrow{Approx'} \subseteq_{pairwise} \overrightarrow{Approx} \Rightarrow F(\overrightarrow{Approx'}) \subseteq_{pairwise} F(\overrightarrow{Approx}) \tag{A.27}$$

It follows from this monotonicity and $(D_2, \subseteq_{pairwise})$ being a complete lattice that F has a unique least fixed point [Tarski, 1955]. This least fixed point shall be referred to as $\overrightarrow{LFP}$.

$$\overrightarrow{LFP} = (\widehat{LFP}^{M_1}, \dots, \widehat{LFP}^{M_m}) \tag{A.28}$$

Note that the initialization of our iterative approach is a member of D_2 and by construction smaller than or equal to this least fixed point.

$$\overrightarrow{Init} \subseteq_{pairwise} \overrightarrow{LFP} \tag{A.29}$$

Moreover, we know due to the monotonicity of the update functions F^{M_a} that whenever a vector of approximation variables is smaller than or equal to the least fixed point, the update of an approximation variable cannot let its value jump above the corresponding component of the least fixed point.

$$\begin{aligned}&\overrightarrow{Approx} \subseteq_{pairwise} \overrightarrow{LFP}\\ \Rightarrow &\forall M_a \in \mathit{Models} : F^{M_a}(\overrightarrow{Approx_{\overline{M_a}}}) \subseteq F^{M_a}(\overrightarrow{LFP_{\overline{M_a}}}) = \widehat{LFP}^{M_a}\end{aligned} \tag{A.30}$$

As a consequence, starting from the initialization, after any sequence of updates of the approximation variables every variable is guaranteed to be a subset of or equal to the corresponding component of the least fixed point of F. This implies that any fixed point reached by our iterative approach starting from the initialization has to coincide with the least fixed point of F.

Now, we are left to show that the least fixed point of F coincides with the least fixed point of equation system (4.37). We show this in two steps.

First we know that the initialization is by design smaller than or equal to every member of D_2.

$$\forall \overrightarrow{Approx} \in D_2 : \overrightarrow{Init} \subseteq_{pairwise} \overrightarrow{Approx} \tag{A.31}$$

Thus, the least fixed point of F is identical the least fixed point of the following function F' on D_2.

$$F' : D_2 \to D_2 \tag{A.32}$$

$$F'(\overrightarrow{Approx}) = (\widehat{Init^{M_1}} \cup F^{M_1}(\overrightarrow{Approx_{\overline{M_1}}}), \ldots, \widehat{Init^{M_m}} \cup F^{M_m}(\overrightarrow{Approx_{\overline{M_m}}})) \tag{A.33}$$

The function F'' extends F' to $D_1 \supseteq D_2$. Note that D_1 is defined in equation (A.11). Further note that the least fixed point of F'' is by design the least fixed point of equation system (4.37).

$$F'' : D_1 \to D_1 \tag{A.34}$$

$$F''(\overrightarrow{Approx}) = (\widehat{Init^{M_1}} \cup F^{M_1}(\overrightarrow{Approx_{\overline{M_1}}}), \ldots, \widehat{Init^{M_m}} \cup F^{M_m}(\overrightarrow{Approx_{\overline{M_m}}})) \tag{A.35}$$

Finally, note that every member of $D_1 \setminus D_2$ is in at least one component smaller than the initialization. Thus, no member of $D_1 \setminus D_2$ can be a fixed point of F''. Consequently, the least fixed point of F'' (i.e. of equation system (4.37)) coincides with the least fixed point of F' and, thus, also with the least fixed point of F. □

Proof of Statement (5.23). We prove statement (5.23) by induction. Consider the following induction hypothesis for $n \in \mathbb{N}$.

$$\{t \in \mathit{Traces} \mid \mathit{len}(t) = n\} \subseteq \{t \mid ((\widehat{t}, \widehat{u}), t) \in \mathit{TraceDescrTrace} \wedge \mathit{len}((\widehat{t}, \widehat{u})) = n\} \tag{A.36}$$

We start the induction by proving that hypothesis (A.36) holds for $n = 0$.

$$\begin{aligned}
&\{t \in \mathit{Traces} \mid \mathit{len}(t) = 0\} \\
\underset{\substack{(5.4)\\(5.5)}}{=}\ &\{t \in \mathit{Traces} \mid \mathit{len}(t) = 0 \wedge t(0) \in S_{init}\} \\
\underset{(5.13)}{\subseteq}\ &\{t \in \mathit{Traces} \mid \mathit{len}(t) = 0 \wedge t(0) \in \gamma(\widehat{s_i}) \wedge \widehat{s_i} \in \widehat{S_{init}}\} \\
\underset{(5.C1)}{\subseteq}\ &\{t \in \mathit{Traces} \mid \mathit{len}(t) = 0 \wedge t(0) \in \gamma(\widehat{t}(0)) \wedge \mathit{len}((\widehat{t}, \widehat{u})) = 0 \wedge (\widehat{t}, \widehat{u}) \in \widehat{\mathit{Traces}}\} \\
\underset{\substack{(5.17)\\(5.18)}}{\subseteq}\ &\{t \in \mathit{Traces} \mid \mathit{len}(t) = 0 \wedge t(0) \in \gamma(\widehat{u}(0)) \wedge \mathit{len}((\widehat{t}, \widehat{u})) = 0 \wedge (\widehat{t}, \widehat{u}) \in \widehat{\mathit{Traces}}\} \\
\underset{\substack{(5.21)\\(5.22)}}{=}\ &\{t \mid ((\widehat{t}, \widehat{u}), t) \in \mathit{TraceDescrTrace} \wedge \mathit{len}((\widehat{t}, \widehat{u})) = 0\}
\end{aligned} \tag{A.37}$$

Subsequently, we show that hypothesis (A.38) holds for $n + 1$ if it holds for n.

$$
\begin{aligned}
& \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1\} \\
\underset{(5.4)}{=}\; & \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1 \wedge (t(n), t(n+1)) \in \mathit{Cycle} \\
& \qquad\qquad \wedge\, t' \in \mathit{Traces} \wedge \mathit{len}(t') = n \\
& \qquad\qquad \wedge\, \forall x \in \mathbb{N}_{\leq n} : t(x) = t'(x)\} \\
\underset{(A.36)}{\subseteq}\; & \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1 \wedge (t(n), t(n+1)) \in \mathit{Cycle} \\
& \qquad\qquad \wedge\, ((\widehat{t'}, \widehat{u'}), t') \in \mathit{TraceDescrTrace} \wedge \mathit{len}((\widehat{t'}, \widehat{u'})) = n \\
& \qquad\qquad \wedge\, \forall x \in \mathbb{N}_{\leq n} : t(x) = t'(x)\} \\
\underset{\substack{(5.21)\\(5.22)}}{=}\; & \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1 \wedge (t(n), t(n+1)) \in \mathit{Cycle} \\
& \qquad\qquad \wedge\, (\widehat{t'}, \widehat{u'}) \in \widehat{\mathit{Traces}} \wedge \mathit{len}((\widehat{t'}, \widehat{u'})) = n \\
& \qquad\qquad \wedge\, t(0) \in \gamma(\widehat{u'}(0)) \wedge \forall x \in \mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n} : t(x) \in \gamma(\widehat{t'}(x))\}
\end{aligned}
$$

$$
\begin{aligned}
\underset{\substack{(5.20)\\(5.18)\\(5.17)}}{=}\; & \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1 \wedge (t(n), t(n+1)) \in \mathit{Cycle} \\
& \qquad\qquad \wedge\, (\widehat{t'}, \widehat{u'}) \in \widehat{\mathit{Traces}} \wedge \mathit{len}((\widehat{t'}, \widehat{u'})) = n \\
& \qquad\qquad \wedge\, [t(0) \in \gamma(\widehat{u'}(0)) \wedge \forall x \in \mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n} : t(x) \in \gamma(\widehat{t'}(x))] \\
& \qquad\qquad \wedge\, t(n) \in \gamma(\widehat{u'}(n))\} \\
\underset{(5.15)}{=}\; & \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1 \wedge (t(n), t(n+1)) \in \mathit{Cycle} \\
& \qquad\qquad \wedge\, (\widehat{t'}, \widehat{u'}) \in \widehat{\mathit{Traces}} \wedge \mathit{len}((\widehat{t'}, \widehat{u'})) = n \\
& \qquad\qquad \wedge\, [t(0) \in \gamma(\widehat{u'}(0)) \wedge \forall x \in \mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n} : t(x) \in \gamma(\widehat{t'}(x))] \\
& \qquad\qquad \wedge\, t(n) \in \gamma(\widehat{u'}(n)) \\
& \qquad\qquad \wedge\, (\widehat{u'}(n), \widehat{s}) \in \widehat{\mathit{Cycle}} \\
& \qquad\qquad \wedge\, t(n+1) \in \gamma(\widehat{s})\} \\
\underset{(5.C2)}{=}\; & \{t \in \mathit{Traces} \mid \mathit{len}(t) = n + 1 \wedge (t(n), t(n+1)) \in \mathit{Cycle} \\
& \qquad\qquad \wedge\, (\widehat{t}, \widehat{u}) \in \widehat{\mathit{Traces}} \wedge \mathit{len}((\widehat{t}, \widehat{u})) = n + 1 \\
& \qquad\qquad \wedge\, [t(0) \in \gamma(\widehat{u}(0)) \wedge \forall x \in \mathbb{N}_{\geq 1} \cap \mathbb{N}_{\leq n+1} : t(x) \in \gamma(\widehat{t}(x))] \\
& \qquad\qquad \wedge\, t(n) \in \gamma(\widehat{u}(n))\} \\
\underset{\substack{(5.21)\\(5.22)\\(5.4)\\(5.17)\\(5.18)}}{=}\; & \{t \mid ((\widehat{t}, \widehat{u}), t) \in \mathit{TraceDescrTrace} \wedge \mathit{len}((\widehat{t}, \widehat{u})) = n + 1\}
\end{aligned}
\tag{A.38}
$$

We inductively follow that hypothesis (A.36) holds for all $n \in \mathbb{N}$. Thus, the following holds as well.

$$
\mathit{Traces} \subseteq \{t \mid ((\widehat{t}, \widehat{u}), t) \in \mathit{TraceDescrTrace}\} \tag{A.39}
$$

As a consequence, statement (5.23) holds. □

Proof of Statement (5.41). We have to show the following statement.

$$\begin{aligned}&\forall(\widehat{t},\widehat{u}) \in \widehat{Traces} : \forall t \in \gamma_{trace}((\widehat{t},\widehat{u})) : \forall x \in \mathbb{N}_{<len((\widehat{t},\widehat{u}))} :\\ &\quad \widehat{E^{LB}}((\widehat{t},\widehat{u}),x) \leq E(t,x) \leq \widehat{E^{UB}}((\widehat{t},\widehat{u}),x)\end{aligned}$$

Note that we replaced $\widehat{t}$ by $(\widehat{t},\widehat{u})$ compared to the original statement (5.41). This was done as this proof needs to access the different components of a member of $\widehat{Traces}$ while this is not necessary in the original statement.

We start by proving the following auxiliary statement.

$$\begin{aligned}&\forall(\widehat{t},\widehat{u}) \in \widehat{Traces} : \forall t \in \gamma_{trace}((\widehat{t},\widehat{u})) : \forall x \in \mathbb{N}_{<len((\widehat{t},\widehat{u}))} :\\ &\quad E(t,x) \Rightarrow \widehat{E^{UB}}((\widehat{t},\widehat{u}),x)\end{aligned} \tag{A.40}$$

Case: $t \in Traces$:

$$\begin{aligned}
& E(t,x)\\
\underset{\substack{(5.10)\\ x<len((\widehat{t},\widehat{u}))\\ (5.32)}}{\Leftrightarrow}\quad & (t(x),t(x+1)) \in E\\
\underset{(5.22)}{\Rightarrow}\quad & [x > 0 \land E \cap (\gamma(\widehat{t}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset] \lor\\
& [x = 0 \land E \cap (\gamma(\widehat{u}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset]\\
\underset{\substack{(5.20)\\ (5.18)\\ (5.17)}}{\Rightarrow}\quad & E \cap (\gamma(\widehat{u}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset\\
\underset{(5.38)}{\Rightarrow}\quad & (\widehat{u}(x),\widehat{t}(x+1)) \in \widehat{E^{UB}}\\
\underset{(5.37)}{\Leftrightarrow}\quad & \widehat{E^{UB}}((\widehat{t},\widehat{u}),x)
\end{aligned}$$

Case: $t = (t',u') \in Spurious$:

$$\begin{aligned}
& E((t',u'),x)\\
\underset{\substack{(5.27)\\ x<len((\widehat{t},\widehat{u}))\\ (5.32)}}{\Leftrightarrow}\quad & (t'(x),u'(x+1)) \in E\\
\underset{(5.29)}{\Rightarrow}\quad & E \cap (\gamma(\widehat{u}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset\\
\underset{(5.38)}{\Rightarrow}\quad & (\widehat{u}(x),\widehat{t}(x+1)) \in \widehat{E^{UB}}\\
\underset{(5.37)}{\Leftrightarrow}\quad & \widehat{E^{UB}}((\widehat{t},\widehat{u}),x)
\end{aligned}$$

This concludes the proof of auxiliary statement (A.40).

Next, we prove a similar auxiliary statement for must-events.

$$\begin{aligned}&\forall(\widehat{t},\widehat{u}) \in \widehat{Traces} : \forall t \in \gamma_{trace}((\widehat{t},\widehat{u})) : \forall x \in \mathbb{N}_{<len((\widehat{t},\widehat{u}))} :\\ &\quad \neg E(t,x) \Rightarrow \neg\widehat{E^{LB}}((\widehat{t},\widehat{u}),x)\end{aligned} \tag{A.41}$$

Case: $t \in \mathit{Traces}$:

$$
\begin{aligned}
& \neg E(t, x) \\
\underset{\substack{(5.10)\\ x<len((\widehat{t},\widehat{u}))\\ (5.32)}}{\Leftrightarrow} \quad & (t(x), t(x+1)) \notin E \\
\underset{(5.22)}{\Rightarrow} \quad & [x > 0 \wedge (\mathit{Cycle} \setminus E) \cap (\gamma(\widehat{t}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset] \vee \\
& [x = 0 \wedge (\mathit{Cycle} \setminus E) \cap (\gamma(\widehat{u}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset] \\
\underset{\substack{(5.20)\\(5.18)\\(5.17)}}{\Rightarrow} \quad & (\mathit{Cycle} \setminus E) \cap (\gamma(\widehat{u}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset \\
\underset{(5.39)}{\Rightarrow} \quad & (\widehat{u}(x), \widehat{t}(x+1)) \notin \widehat{E^{LB}} \\
\underset{(5.37)}{\Leftrightarrow} \quad & \neg \widehat{E^{LB}}((\widehat{t}, \widehat{u}), x)
\end{aligned}
$$

Case: $t = (t', u') \in \mathit{Spurious}$:

$$
\begin{aligned}
& \neg E((t', u'), x) \\
\underset{\substack{(5.27)\\ x<len((\widehat{t},\widehat{u}))\\ (5.32)}}{\Leftrightarrow} \quad & (t'(x), u'(x+1)) \notin E \\
\underset{(5.29)}{\Rightarrow} \quad & (\mathit{Cycle} \setminus E) \cap (\gamma(\widehat{u}(x)) \times \gamma(\widehat{t}(x+1))) \neq \emptyset \\
\underset{(5.39)}{\Rightarrow} \quad & (\widehat{u}(x), \widehat{t}(x+1)) \notin \widehat{E^{LB}} \\
\underset{(5.37)}{\Leftrightarrow} \quad & \neg \widehat{E^{LB}}((\widehat{t}, \widehat{u}), x)
\end{aligned}
$$

This concludes the proof of auxiliary statement (A.41).

Statements (A.40) and (A.41) together imply the statement we originally wanted to prove.

$$
\widehat{E^{LB}}((\widehat{t}, \widehat{u}), x) \underset{(A.41)}{\leq} E(t, x) \underset{(A.40)}{\leq} \widehat{E^{UB}}((\widehat{t}, \widehat{u}), x) \qquad \square
$$

Proof of the Soundness of the Lifting Rules in Table 5.1. We prove the soundness of the property lifting rules by structural induction on the different logical constructs that may appear in the system properties we consider. In particular, we prove the two following hypotheses for each of the logical constructs. When doing so, we rely on both hypotheses holding for the sub-properties that the logical constructs are build of.

$$
[\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \Rightarrow \boldsymbol{lift(}P(t, \vec{x})\boldsymbol{)} \tag{A.H1}
$$

$$
[\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \Rightarrow \neg \boldsymbol{flip(lift(}P(t, \vec{x})\boldsymbol{))} \tag{A.H2}
$$

Hypothesis (A.H1) is the actual soundness hypothesis. For top-level properties—which do not argue about free variables (i.e. vector $\vec{x}$ does not exist)—it is identical to the soundness criterion (4.C1) for lifted properties. The additional hypothesis (A.H2) is required to show hypothesis (A.H1) for some of the logical constructs.

Now, we present the induction step for each of the first seven lifting rules in Table 5.1. The last lifting rule—for logical implication—can be derived as a combination of the other lifting rules and, thus, is omitted in the induction proof. We present its derivation in the end.

1. $P(t, \vec{x}) \Leftrightarrow E(t, x_i)$

$$\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : E(t, x_i)] \\
\underset{(5.41)}{\Rightarrow}\ & \widehat{E^{UB}}(\widehat{t}, x_i) \\
\underset{(\text{LR1})}{\Leftrightarrow}\ & \boldsymbol{lift(}E(t, x_i)\boldsymbol{)} \\
\Leftrightarrow\ & \boldsymbol{lift(}P(t, \vec{x})\boldsymbol{)}
\end{aligned}$$

$$\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg E(t, x_i)] \\
\underset{(5.41)}{\Rightarrow}\ & \neg\widehat{E^{LB}}(\widehat{t}, x_i) \\
\underset{(\text{LR9})}{\Leftrightarrow}\ & \neg\boldsymbol{flip(}\widehat{E^{UB}}(\widehat{t}, x_i)\boldsymbol{)} \\
\underset{(\text{LR1})}{\Leftrightarrow}\ & \neg\boldsymbol{flip(lift(}E(t, x_i)\boldsymbol{))} \\
\Leftrightarrow\ & \neg\boldsymbol{flip(lift(}P(t, \vec{x})\boldsymbol{))}
\end{aligned}$$

2. $P(t, \vec{x}) \Leftrightarrow [P_1(t, \vec{x}) \wedge P_2(t, \vec{x})]$

$$\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x}) \wedge P_2(t, \vec{x})] \\
\Rightarrow\ & [(\exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x})) \wedge (\exists t \in \gamma_{trace}(\widehat{t}) : P_2(t, \vec{x}))] \\
\underset{(\text{A.H1})}{\Rightarrow}\ & [\boldsymbol{lift(}P_1(t, \vec{x})\boldsymbol{)} \wedge \boldsymbol{lift(}P_2(t, \vec{x})\boldsymbol{)}] \\
\underset{(\text{LR2})}{\Leftrightarrow}\ & \boldsymbol{lift(}[P_1(t, \vec{x}) \wedge P_2(t, \vec{x})]\boldsymbol{)} \\
\Leftrightarrow\ & \boldsymbol{lift(}P(t, \vec{x})\boldsymbol{)}
\end{aligned}$$

$$\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow \quad & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg(P_1(t, \vec{x}) \wedge P_2(t, \vec{x}))] \\
\Leftrightarrow \quad & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x}) \vee \neg P_2(t, \vec{x})] \\
\Leftrightarrow \quad & [(\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x})) \vee (\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_2(t, \vec{x}))] \\
\underset{\text{(A.H2)}}{\Rightarrow} \quad & [\neg \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}))) \vee \neg \boldsymbol{flip}(\boldsymbol{lift}(P_2(t, \vec{x})))] \\
\Leftrightarrow \quad & \neg[\boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}))) \wedge \boldsymbol{flip}(\boldsymbol{lift}(P_2(t, \vec{x})))] \\
\underset{\text{(LR11)}}{\Leftrightarrow} \quad & \neg \boldsymbol{flip}([\boldsymbol{lift}(P_1(t, \vec{x})) \wedge \boldsymbol{lift}(P_2(t, \vec{x}))]) \\
\underset{\text{(LR2)}}{\Leftrightarrow} \quad & \neg \boldsymbol{flip}(\boldsymbol{lift}([P_1(t, \vec{x}) \wedge P_2(t, \vec{x})])) \\
\Leftrightarrow \quad & \neg \boldsymbol{flip}(\boldsymbol{lift}(P(t, \vec{x})))
\end{aligned}$$

3. $P(t, \vec{x}) \Leftrightarrow [P_1(t, \vec{x}) \vee P_2(t, \vec{x})]$

$$\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow \quad & [\exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x}) \vee P_2(t, \vec{x})] \\
\Leftrightarrow \quad & [(\exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x})) \vee (\exists t \in \gamma_{trace}(\widehat{t}) : P_2(t, \vec{x}))] \\
\underset{\text{(A.H1)}}{\Rightarrow} \quad & [\boldsymbol{lift}(P_1(t, \vec{x})) \vee \boldsymbol{lift}(P_2(t, \vec{x}))] \\
\underset{\text{(LR3)}}{\Leftrightarrow} \quad & \boldsymbol{lift}([P_1(t, \vec{x}) \vee P_2(t, \vec{x})]) \\
\Leftrightarrow \quad & \boldsymbol{lift}(P(t, \vec{x}))
\end{aligned}$$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow\quad & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg(P_1(t, \vec{x}) \vee P_2(t, \vec{x}))] \\
\Leftrightarrow\quad & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x}) \wedge \neg P_2(t, \vec{x})] \\
\Rightarrow\quad & [(\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x})) \wedge (\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_2(t, \vec{x}))] \\
\underset{\text{(A.H2)}}{\Rightarrow}\quad & [\neg \boldsymbol{flip(lift(}P_1(t, \vec{x})\boldsymbol{))} \wedge \neg \boldsymbol{flip(lift(}P_2(t, \vec{x})\boldsymbol{))}] \\
\Leftrightarrow\quad & \neg[\boldsymbol{flip(lift(}P_1(t, \vec{x})\boldsymbol{))} \vee \boldsymbol{flip(lift(}P_2(t, \vec{x})\boldsymbol{))}] \\
\underset{\text{(LR12)}}{\Leftrightarrow}\quad & \neg \boldsymbol{flip(}[\boldsymbol{lift(}P_1(t, \vec{x})\boldsymbol{)} \vee \boldsymbol{lift(}P_2(t, \vec{x})\boldsymbol{)}]\boldsymbol{)} \\
\underset{\text{(LR3)}}{\Leftrightarrow}\quad & \neg \boldsymbol{flip(lift(}[P_1(t, \vec{x}) \vee P_2(t, \vec{x})]\boldsymbol{))} \\
\Leftrightarrow\quad & \neg \boldsymbol{flip(lift(}P(t, \vec{x})\boldsymbol{))}
\end{aligned}
$$

4. $P(t, \vec{x}) \Leftrightarrow [\ \forall x \in X : P_1(t, \vec{x}, x)\ ]$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow\quad & [\exists t \in \gamma_{trace}(\widehat{t}) : \forall x \in X : P_1(t, \vec{x}, x)] \\
\Rightarrow\quad & [\forall x \in X : \exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x}, x)] \\
\underset{\text{(A.H1)}}{\Rightarrow}\quad & [\forall x \in X : \boldsymbol{lift(}P_1(t, \vec{x}, x)\boldsymbol{)}] \\
\underset{\text{(LR4)}}{\Leftrightarrow}\quad & \boldsymbol{lift(}[\forall x \in X : P_1(t, \vec{x}, x)]\boldsymbol{)} \\
\Leftrightarrow\quad & \boldsymbol{lift(}P(t, \vec{x})\boldsymbol{)}
\end{aligned}
$$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg \forall x \in X : P_1(t, \vec{x}, x)] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \exists x \in X : \neg P_1(t, \vec{x}, x)] \\
\Leftrightarrow\ & [\exists x \in X : \exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x}, x)] \\
\underset{\text{(A.H2)}}{\Rightarrow}\ & [\exists x \in X : \neg \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}, x)))] \\
\Leftrightarrow\ & \neg[\forall x \in X : \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}, x)))] \\
\underset{\text{(LR13)}}{\Leftrightarrow}\ & \neg \boldsymbol{flip}([\forall x \in X : \boldsymbol{lift}(P_1(t, \vec{x}, x))]) \\
\underset{\text{(LR4)}}{\Leftrightarrow}\ & \neg \boldsymbol{flip}(\boldsymbol{lift}([\forall x \in X : P_1(t, \vec{x}, x)])) \\
\Leftrightarrow\ & \neg \boldsymbol{flip}(\boldsymbol{lift}(P(t, \vec{x})))
\end{aligned}
$$

5. $P(t, \vec{x}) \Leftrightarrow [\ \exists x \in X : P_1(t, \vec{x}, x)\ ]$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \exists x \in X : P_1(t, \vec{x}, x)] \\
\Leftrightarrow\ & [\exists x \in X : \exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x}, x)] \\
\underset{\text{(A.H1)}}{\Rightarrow}\ & [\exists x \in X : \boldsymbol{lift}(P_1(t, \vec{x}, x))] \\
\underset{\text{(LR5)}}{\Leftrightarrow}\ & \boldsymbol{lift}([\exists x \in X : P_1(t, \vec{x}, x)]) \\
\Leftrightarrow\ & \boldsymbol{lift}(P(t, \vec{x}))
\end{aligned}
$$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg \exists x \in X : P_1(t, \vec{x}, x)] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \forall x \in X : \neg P_1(t, \vec{x}, x)] \\
\Rightarrow\ & [\forall x \in X : \exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x}, x)] \\
\underset{\text{(A.H2)}}{\Rightarrow}\ & [\forall x \in X : \neg \boldsymbol{flip(lift(}P_1(t, \vec{x}, x)\boldsymbol{))}] \\
\Leftrightarrow\ & \neg[\exists x \in X : \boldsymbol{flip(lift(}P_1(t, \vec{x}, x)\boldsymbol{))}] \\
\underset{\text{(LR14)}}{\Leftrightarrow}\ & \neg \boldsymbol{flip(}[\exists x \in X : \boldsymbol{lift(}P_1(t, \vec{x}, x)\boldsymbol{)}]\boldsymbol{)} \\
\underset{\text{(LR5)}}{\Leftrightarrow}\ & \neg \boldsymbol{flip(lift(}[\exists x \in X : P_1(t, \vec{x}, x)]\boldsymbol{))} \\
\Leftrightarrow\ & \neg \boldsymbol{flip(lift(}P(t, \vec{x})\boldsymbol{))}
\end{aligned}
$$

6. $P(t, \vec{x}) \Leftrightarrow \neg P_1(t, \vec{x})$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P_1(t, \vec{x})] \\
\underset{\text{(A.H2)}}{\Rightarrow}\ & \neg \boldsymbol{flip(lift(}P_1(t, \vec{x})\boldsymbol{))} \\
\underset{\text{(LR6)}}{\Leftrightarrow}\ & \boldsymbol{lift(}\neg P_1(t, \vec{x})\boldsymbol{)} \\
\Leftrightarrow\ & \boldsymbol{lift(}P(t, \vec{x})\boldsymbol{)}
\end{aligned}
$$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg\neg P_1(t, \vec{x})] \\
\Leftrightarrow\ & [\exists t \in \gamma_{trace}(\widehat{t}) : P_1(t, \vec{x})] \\
\underset{\text{(A.H1)}}{\Rightarrow}\ & \boldsymbol{lift(}P_1(t, \vec{x})\boldsymbol{)} \\
\underset{\text{(LR18)}}{\Leftrightarrow}\ & \boldsymbol{flip(flip(lift(}P_1(t, \vec{x})\boldsymbol{)))} \\
\Leftrightarrow\ & \neg\neg \boldsymbol{flip(flip(lift(}P_1(t, \vec{x})\boldsymbol{)))} \\
\underset{\text{(LR15)}}{\Leftrightarrow}\ & \neg \boldsymbol{flip(}\neg \boldsymbol{flip(lift(}P_1(t, \vec{x})\boldsymbol{)))} \\
\underset{\text{(LR6)}}{\Leftrightarrow}\ & \neg \boldsymbol{flip(lift(}\neg P_1(t, \vec{x})\boldsymbol{))} \\
\Leftrightarrow\ & \neg \boldsymbol{flip(lift(}P(t, \vec{x})\boldsymbol{))}
\end{aligned}
$$

7. $P(t, \vec{x}) \Leftrightarrow [\ a_1 \cdot \sum_{x \in X} P_1(t, \vec{x}, x) + b_1 \lesssim a_2 \cdot \sum_{y \in Y} P_2(t, \vec{x}, y) + b_2\ ]$

   In order to prove the hypotheses for the case of a generic inequation, we derive two auxiliary statements from the hypotheses. Hypothesis (A.H1) implies that the value of a lifted property for a sequence of abstract states is always at least as high as the value of the corresponding concrete property for any described trace.

$$\forall t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x}) \leq \boldsymbol{lift}(P(t, \vec{x})) \tag{A.42}$$

   Similarly, hypothesis (A.H2) implies that the value of the flipped version of a lifted property for a sequence of abstract states does never exceed the value of the corresponding concrete property for any described trace.

$$\forall t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x}) \geq \boldsymbol{flip}(\boldsymbol{lift}(P(t, \vec{x}))) \tag{A.43}$$

   Now, we prove the hypotheses also for the generic inequation.

$$\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : P(t, \vec{x})] \\
\Leftrightarrow\quad & [\exists t \in \gamma_{trace}(\widehat{t}) : a_1 \cdot \sum_{x \in X} P_1(t, \vec{x}, x) + b_1 \lesssim a_2 \cdot \sum_{y \in Y} P_2(t, \vec{x}, y) + b_2] \\
\underset{\substack{(A.42)\\(A.43)\\a_1 \geq 0\\a_2 \geq 0\\\sum monotone}}{\Rightarrow}\quad & [a_1 \cdot \sum_{x \in X} \boldsymbol{flip}(\boldsymbol{lift}(P_1(t, \vec{x}, x))) + b_1 \lesssim a_2 \cdot \sum_{y \in Y} \boldsymbol{lift}(P_2(t, \vec{x}, y)) + b_2] \\
\underset{(LR7)}{\Leftrightarrow}\quad & \boldsymbol{lift}([a_1 \cdot \sum_{x \in X} P_1(t, \vec{x}, x) + b_1 \lesssim a_2 \cdot \sum_{y \in Y} P_2(t, \vec{x}, y) + b_2]) \\
\Leftrightarrow\quad & \boldsymbol{lift}(P(t, \vec{x}))
\end{aligned}$$

$$
\begin{aligned}
& [\exists t \in \gamma_{trace}(\widehat{t}) : \neg P(t, \vec{x})] \\
\Leftrightarrow\quad & [\exists t \in \gamma_{trace}(\widehat{t}) : \neg(a_1 \cdot \sum_{x\in X} P_1(t,\vec{x},x) + b_1 \lesssim a_2 \cdot \sum_{y\in Y} P_2(t,\vec{x},y) + b_2)] \\
\Leftrightarrow\quad & [\exists t \in \gamma_{trace}(\widehat{t}) : a_1 \cdot \sum_{x\in X} P_1(t,\vec{x},x) + b_1 \not\lesssim a_2 \cdot \sum_{y\in Y} P_2(t,\vec{x},y) + b_2] \\
\underset{\substack{(A.42)\\(A.43)\\a_1\geq 0\\a_2\geq 0\\\sum monotone}}{\Rightarrow}\quad & [a_1 \cdot \sum_{x\in X} \boldsymbol{lift}(P_1(t,\vec{x},x)) + b_1 \not\lesssim a_2 \cdot \sum_{y\in Y} \boldsymbol{flip}(\boldsymbol{lift}(P_2(t,\vec{x},y))) + b_2] \\
\Leftrightarrow\quad & \neg[a_1 \cdot \sum_{x\in X} \boldsymbol{lift}(P_1(t,\vec{x},x)) + b_1 \lesssim a_2 \cdot \sum_{y\in Y} \boldsymbol{flip}(\boldsymbol{lift}(P_2(t,\vec{x},y))) + b_2] \\
\underset{(LR18)}{\Leftrightarrow}\quad & \neg\boldsymbol{flip}(\boldsymbol{flip}([a_1 \cdot \sum_{x\in X} \boldsymbol{lift}(P_1(t,\vec{x},x)) + b_1 \lesssim a_2 \cdot \sum_{y\in Y} \boldsymbol{flip}(\boldsymbol{lift}(P_2(t,\vec{x},y))) + b_2])) \\
\underset{(LR16)}{\Leftrightarrow}\quad & \neg\boldsymbol{flip}([a_1 \cdot \sum_{x\in X} \boldsymbol{flip}(\boldsymbol{lift}(P_1(t,\vec{x},x))) + b_1 \\
& \qquad \lesssim a_2 \cdot \sum_{y\in Y} \boldsymbol{flip}(\boldsymbol{flip}(\boldsymbol{lift}(P_2(t,\vec{x},y)))) + b_2]) \\
\underset{(LR18)}{\Leftrightarrow}\quad & \neg\boldsymbol{flip}([a_1 \cdot \sum_{x\in X} \boldsymbol{flip}(\boldsymbol{lift}(P_1(t,\vec{x},x))) + b_1 \lesssim a_2 \cdot \sum_{y\in Y} \boldsymbol{lift}(P_2(t,\vec{x},y)) + b_2]) \\
\underset{(LR7)}{\Leftrightarrow}\quad & \neg\boldsymbol{flip}(\boldsymbol{lift}([a_1 \cdot \sum_{x\in X} P_1(t,\vec{x},x) + b_1 \lesssim a_2 \cdot \sum_{y\in Y} P_2(t,\vec{x},y) + b_2])) \\
\Leftrightarrow\quad & \neg\boldsymbol{flip}(\boldsymbol{lift}(P(t,\vec{x})))
\end{aligned}
$$

This concludes the inductive proof of the first seven lifting rules in Table 5.1. Finally, the last lifting rule in Table 5.1 is derived by applying the other rules:

$$
\begin{aligned}
& \boldsymbol{lift}([P_1(t,\vec{x},x) \Rightarrow P_2(t,\vec{x},x)]) \\
\Leftrightarrow\quad & \boldsymbol{lift}([\neg P_1(t,\vec{x},x) \vee P_2(t,\vec{x},x)]) \\
\underset{(LR3)}{\Leftrightarrow}\quad & [\boldsymbol{lift}(\neg P_1(t,\vec{x},x)) \vee \boldsymbol{lift}(P_2(t,\vec{x},x))] \\
\underset{(LR6)}{\Leftrightarrow}\quad & [\neg\boldsymbol{flip}(\boldsymbol{lift}(P_1(t,\vec{x},x))) \vee \boldsymbol{lift}(P_2(t,\vec{x},x))] \\
\Leftrightarrow\quad & [\boldsymbol{flip}(\boldsymbol{lift}(P_1(t,\vec{x},x))) \Rightarrow \boldsymbol{lift}(P_2(t,\vec{x},x))] \quad \Box
\end{aligned}
$$

Soundness of the Lifting Rules in Table 5.3. This proof has to show that the lifting rules in Table 5.3 fulfill criterion (5.C4). We do not prove this directly as we do not want to argue about the edges and nodes of graphs in our proof. Instead we argue about helper sets that overapproximate the paths through arbitrary graphs and the things described by these paths.

First, we define a helper set $\widehat{hPaths}$. Essentially, it is an overapproximation of the paths that might result from arbitrary graphs. As such, we do no longer define it with respect to the nodes and edges of an actual graph.

$$\widehat{hPaths} = \{\widehat{p} : \mathbb{N}_{<n} \times \mathit{Events} \times \{\mathit{UB}, \mathit{LB}\} \to \mathbb{N} \mid n \in \mathbb{N}\} \tag{A.44}$$

We provide the same helper functions that are already available for paths.

$$\forall n \in \mathbb{N} : \forall \widehat{p} \in \widehat{hPaths} \cap (\mathbb{N}_{<n} \times \mathit{Events} \times \{\mathit{UB}, \mathit{LB}\} \to \mathbb{N}) : \mathit{len}(\widehat{p}) = n \tag{A.45}$$

$$\widehat{wE^{UB}}, \widehat{wE^{LB}} : \bigcup_{\widehat{p} \in \widehat{hPaths}} (\{\widehat{p}\} \times \mathbb{N}_{<\mathit{len}(\widehat{p})}) \to \mathbb{N} \tag{A.46}$$

$$\widehat{wE^{UB}}(\widehat{p}, x) \equiv \widehat{p}(x, E, \mathit{UB}) \tag{A.47}$$

$$\widehat{wE^{LB}}(\widehat{p}, x) \equiv \widehat{p}(x, E, \mathit{LB}) \tag{A.48}$$

Since, in the definition of set $\widehat{hPaths}$, there are no further restrictions on the members of the set, any subpath of our actual graph has a corresponding member in set $\widehat{hPaths}$.

$$\begin{aligned}
&\forall \widehat{p} \in \widehat{SubPaths} : \exists \widehat{p'} \in \widehat{hPaths} : \\
&\quad \mathit{len}(\widehat{p'}) = \mathit{len}(\widehat{p}) \land \\
&\quad \forall x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} : \\
&\qquad \widehat{wE^{UB}}(\widehat{p'}, x) = \widehat{wE^{UB}}(\widehat{p}, x) \land \\
&\qquad \widehat{wE^{LB}}(\widehat{p'}, x) = \widehat{wE^{LB}}(\widehat{p}, x)
\end{aligned} \tag{A.49}$$

The properties that we specify on paths only argue about the length of a path and the event-bounding weights at the different positions of a path. As a consequence, any of these properties evaluated on a particular path evaluates to the same truth value as when evaluated on the corresponding member of set $\widehat{hPaths}$.

Analogously, we define a helper set $\widehat{hTraces}$.

$$\widehat{hTraces} = \{\widehat{t} : \mathbb{N}_{<n} \times \mathit{Events} \times \{\mathit{UB}, \mathit{LB}\} \to \mathbb{N} \mid n \in \mathbb{N}\} \tag{A.50}$$

For its members, we define the same helper functions that are available for the members of $\widehat{Traces}$ and $\widehat{SpuriousTraces}$.

$$\forall n \in \mathbb{N} : \forall \widehat{t} \in \widehat{hTraces} \cap (\mathbb{N}_{<n} \times \mathit{Events} \times \{\mathit{UB}, \mathit{LB}\} \to \mathbb{N}) : \mathit{len}(\widehat{t}) = n \tag{A.51}$$

$$\widehat{E^{UB}}, \widehat{E^{LB}} : \bigcup_{\widehat{t} \in \widehat{hTraces}} (\{\widehat{t}\} \times \mathbb{N}_{<\mathit{len}(\widehat{t})}) \to \mathbb{N} \tag{A.52}$$

$$\widehat{E^{UB}}(\widehat{t}, x) \equiv \widehat{t}(x, E, \mathit{UB}) \tag{A.53}$$

$$\widehat{E^{LB}}(\widehat{t}, x) \equiv \widehat{t}(x, E, \mathit{LB}) \tag{A.54}$$

Since, in the definition of set $\widehat{hTraces}$, there are no further restrictions on the members of the set, any of $\widehat{Traces}$ or $\widehat{SpuriousTraces}$ has a corresponding member in set $\widehat{hTraces}$.

$$\begin{aligned}
&\forall \widehat{t} \in \widehat{Traces} \cup \widehat{SpuriousTraces} : \exists \widehat{t'} \in \widehat{hTraces} : \\
&\quad len(\widehat{t'}) = len(\widehat{t}) \wedge \\
&\quad \forall x \in \mathbb{N}_{<len(\widehat{t})} : \\
&\qquad \widehat{E^{UB}}(\widehat{t'}, x) = \widehat{E^{UB}}(\widehat{t}, x) \wedge \\
&\qquad \widehat{E^{LB}}(\widehat{t'}, x) = \widehat{E^{LB}}(\widehat{t}, x)
\end{aligned} \tag{A.55}$$

The properties that we specify on members of $\widehat{Traces}$ or $\widehat{SpuriousTraces}$ only argue about their length and/or event bounds. As a consequence, any of these properties evaluated on a particular member of $\widehat{Traces}$ or $\widehat{SpuriousTraces}$ evaluates to the same truth value as when evaluated on the corresponding member of set $\widehat{hTraces}$.

Next, we connect the members of $\widehat{hPaths}$ and $\widehat{hTraces}$ by a description relation that is defined analogously to the relations *PathDescrTrace* and *PathDescrSpuriousTrace* introduced earlier.

$$hPathDescrHTrace \subseteq \widehat{hPaths} \times \widehat{hTraces} \tag{A.56}$$

$$\begin{aligned}
&(\widehat{p}, \widehat{t}) \in hPathDescrHTrace \\
\Leftrightarrow &\exists part \in Partitionings(len(\widehat{p}), len(\widehat{t})) : \\
&\quad \forall E \in Events : \\
&\qquad \forall x \in \mathbb{N}_{<len(\widehat{p})} : \\
&\qquad\quad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{E^{UB}}(\widehat{t}, i) \leq \widehat{wE^{UB}}(\widehat{p}, x) \wedge \\
&\qquad\quad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{E^{LB}}(\widehat{t}, i) \geq \widehat{wE^{LB}}(\widehat{p}, x)
\end{aligned} \tag{A.57}$$

From equations (A.49), (A.55), and their consequences on the kinds of properties we consider we follow an additional auxiliary statement about the relation just defined.

$$\begin{aligned}
&\exists \widehat{p} \in \widehat{SubPaths} : \exists \widehat{t} \in \gamma_{path}(\widehat{p}) : \widehat{P_A}(\widehat{t}) \wedge \widehat{P_B}(\widehat{p}) \\
\Rightarrow &\exists \widehat{p} \in \widehat{hPaths} : \exists \widehat{t} \in \widehat{hTraces} : (\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_A}(\widehat{t}) \wedge \widehat{P_B}(\widehat{p})
\end{aligned} \tag{A.58}$$

Finally, we take a look at an implication of criterion (5.C4) not holding.

$$\begin{aligned}
&\neg(5.C4) \\
\underset{(5.C4)}{\Leftrightarrow} \; &\neg[\forall \widehat{p} \in \widehat{SubPaths} : [\, \exists \widehat{t} \in \gamma_{path}(\widehat{p}) : \widehat{P_k}(\widehat{t}) \,] \Rightarrow \widehat{P_k^{path}}(\widehat{p})] \\
\Leftrightarrow \; &\exists \widehat{p} \in \widehat{SubPaths} : [\, \exists \widehat{t} \in \gamma_{path}(\widehat{p}) : \widehat{P_k}(\widehat{t}) \,] \wedge \neg\widehat{P_k^{path}}(\widehat{p}) \\
\Leftrightarrow \; &\exists \widehat{p} \in \widehat{SubPaths} : \exists \widehat{t} \in \gamma_{path}(\widehat{p}) : \widehat{P_k}(\widehat{t}) \wedge \neg\widehat{P_k^{path}}(\widehat{p}) \\
\underset{(A.58)}{\Rightarrow} \; &\exists \widehat{p} \in \widehat{hPaths} : \exists \widehat{t} \in \widehat{hTraces} : (\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_k}(\widehat{t}) \wedge \neg\widehat{P_k^{path}}(\widehat{p})
\end{aligned}$$

By contraposition, we obtain the following, slightly stronger criterion that implies criterion (5.C4).

$$\begin{aligned}&\neg\exists \widehat{p} \in \widehat{hPaths} : \exists \widehat{t} \in \widehat{hTraces} : (\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_k}(\widehat{t}) \wedge \neg \widehat{P_k^{path}}(\widehat{p}) \\ &\Rightarrow (5.C4)\end{aligned} \tag{A.59}$$

Thus, we are left to show that there are no pair $(\widehat{p}, \widehat{t})$ in the cross product of our helper sets such that the following statement is fulfilled.

$$(\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_k}(\widehat{t}) \wedge \neg \widehat{P_k^{path}}(\widehat{p})$$

For some properties we might be able to automatically show the unsatisfiability of a corresponding SMT (satisfyability modulo theory) problem. Unfortunately, our experiments showed that a current version of the Z3 SMT solver is not able to show the unsatisfiability of the SMT problem formulations for the lifting rules in Table 5.3 within a runtime of one day. Nonetheless, this might be an attractive option for proving the soundness of future lifting rules provided that either the properties involved in them are very simple or the SMT community can significantly advance the state of the art in SMT solving.

For now, we resort to a manual proof of the left-hand side of the implication in equation (A.59). To this end, we present the left-hand side as a further implication in an equivalent way.

$$\begin{aligned}&[\forall \widehat{p} \in \widehat{hPaths} : \forall \widehat{t} \in \widehat{hTraces} : (\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_k}(\widehat{t}) \Rightarrow \widehat{P_k^{path}}(\widehat{p})] \\ &\Rightarrow (5.C4)\end{aligned}$$

Thus, we are left to show that for every pair $(\widehat{p}, \widehat{t})$ in the cross product of our helper sets the following implication is fulfilled.

$$(\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_k}(\widehat{t}) \Rightarrow \widehat{P_k^{path}}(\widehat{p})$$

Now, we prove that this implication holds for each lifting rule in Table 5.3.

1. $\widehat{P_k}(\widehat{t}) \Leftrightarrow \forall z \in \mathbb{N}_{\leq len(\widehat{t})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2$

$$
\begin{aligned}
& (\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \widehat{P_k}(\widehat{t}) \\
\Leftrightarrow\ & (\widehat{p}, \widehat{t}) \in hPathDescrHTrace \wedge \\
& \quad \forall z \in \mathbb{N}_{\leq len(\widehat{t})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2 \\
\underset{(A.57)}{\Leftrightarrow}\ & [\exists part \in Partitionings(len(\widehat{p}), len(\widehat{t})) : \\
& \quad\quad \forall E \in Events : \\
& \quad\quad\quad \forall x \in \mathbb{N}_{<len(\widehat{p})} : \\
& \quad\quad\quad\quad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{E^{UB}}(\widehat{t}, i) \leq \widehat{wE^{UB}}(\widehat{p}, x) \wedge \\
& \quad\quad\quad\quad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{E^{LB}}(\widehat{t}, i) \geq \widehat{wE^{LB}}(\widehat{p}, x) \\
&] \wedge \\
& \quad \forall z \in \mathbb{N}_{\leq len(\widehat{t})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2
\end{aligned}
$$

$$\Rightarrow \quad [\exists part \in Partitionings(len(\widehat{p}), len(\widehat{t})) :$$

$$\forall x \in \mathbb{N}_{<len(\widehat{p})} :$$

$$\sum_{from(part,x) \leq i \leq to(part,x)} \widehat{E_2^{UB}}(\widehat{t}, i) \leq \widehat{wE_2^{UB}}(\widehat{p}, x) \wedge$$

$$\sum_{from(part,x) \leq i \leq to(part,x)} \widehat{E_1^{LB}}(\widehat{t}, i) \geq \widehat{wE_1^{LB}}(\widehat{p}, x)$$

$$] \wedge$$

$$\forall z \in \mathbb{N}_{\leq len(\widehat{t})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2$$

$$\Rightarrow \quad [\exists part \in Partitionings(len(\widehat{p}), len(\widehat{t})) :$$

$$\forall x \in \mathbb{N}_{<len(\widehat{p})} :$$

$$\sum_{0 \leq i \leq to(part,x)} \widehat{E_2^{UB}}(\widehat{t}, i) \leq \sum_{0 \leq y \leq x} \widehat{wE_2^{UB}}(\widehat{p}, y) \wedge$$

$$\sum_{0 \leq i \leq to(part,x)} \widehat{E_1^{LB}}(\widehat{t}, i) \geq \sum_{0 \leq y \leq x} \widehat{wE_1^{LB}}(\widehat{p}, y)$$

$$] \wedge$$

$$\forall z \in \mathbb{N}_{\leq len(\widehat{t})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2$$

$$\underset{\substack{a_1 \geq 0 \\ a_2 \geq 0 \\ \sum monotone}}{\Rightarrow} \quad [\exists part \in Partitionings(len(\widehat{p}), len(\widehat{t})) :$$

$$\forall x \in \mathbb{N}_{<len(\widehat{p})} :$$

$$a_2 \cdot \sum_{0 \leq i \leq to(part,x)} \widehat{E_2^{UB}}(\widehat{t}, i) + b_2 \leq a_2 \cdot \sum_{0 \leq y \leq x} \widehat{wE_2^{UB}}(\widehat{p}, y) + b_2 \wedge$$

$$a_1 \cdot \sum_{0 \leq i \leq to(part,x)} \widehat{E_1^{LB}}(\widehat{t}, i) + b_1 \geq a_1 \cdot \sum_{0 \leq y \leq x} \widehat{wE_1^{LB}}(\widehat{p}, y) + b_1$$

$$] \wedge$$

$$\forall z \in \mathbb{N}_{\leq len(\widehat{t})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_1^{LB}}(\widehat{t}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{E_2^{UB}}(\widehat{t}, x) + b_2$$

$$\underset{\lesssim \in \{<, \leq\}}{\Rightarrow} \quad \forall z \in \mathbb{N}_{\leq len(\widehat{p})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{wE_1^{LB}}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{wE_2^{UB}}(\widehat{p}, x) + b_2$$

$$\underset{\text{(LR19)}}{\Leftrightarrow} \quad \boldsymbol{lift}(\widehat{P_k}(\widehat{t}))$$

This concludes the proof of the lifting rules in Table 5.3. □

Proof of Statement 5.84. Essentially, we have to show the following:

$$\forall \widehat{p} \in \widehat{Paths} : \exists \widehat{i} \in \widehat{Implicit} : (\widehat{i}, \widehat{p}) \in \mathit{ImplicitDescrPath} \tag{A.60}$$

Using the definition of $\widehat{Implicit}$ (equation (5.81)), we can expand this to the following equivalent statement.

$$\begin{aligned}
&\forall \widehat{p} \in \widehat{Paths} : \\
&\quad \exists (\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd}) \in (\mathit{Edges} \to \mathbb{N}) \times (\mathit{Edges} \to \{0,1\})^2 : \\
&\qquad [\ \forall e \in \mathit{Edges} : \mathit{isStart}(e) \leq \mathit{timesTaken}(e)\ ] \land \\
&\qquad [\ \forall e \in \mathit{Edges} : \mathit{isEnd}(e) \leq \mathit{timesTaken}(e)\ ] \land \\
&\qquad \sum_{e \in \mathit{Edges}} \mathit{isStart}(e) \leq 1 \land \\
&\qquad \sum_{e \in \mathit{Edges}} \mathit{isEnd}(e) \leq 1 \land \\
&\qquad \sum_{e \in \mathit{Edges}} \mathit{isStart}(e) = \sum_{e \in \mathit{Edges}} \mathit{isEnd}(e) \land \\
&\qquad \sum_{e \in (\mathit{Edges} \setminus \mathit{Edges}_{start})} \mathit{isStart}(e) = 0 \land \\
&\qquad \sum_{e \in (\mathit{Edges} \setminus \mathit{Edges}_{end})} \mathit{isEnd}(e) = 0 \land \\
&\qquad [\ \mathit{Nodes}_{start} \cap \mathit{Nodes}_{end} = \emptyset \Rightarrow \sum_{e \in \mathit{Edges}} \mathit{isStart}(e) = 1\ ] \land \\
&\qquad [\ \forall \mathit{node} \in \mathit{Nodes} : \\
&\qquad\qquad \sum_{e_{in} \in \mathit{inEdges}(\mathit{node})} [\mathit{timesTaken}(e_{in}) - \mathit{isEnd}(e_{in})] \\
&\qquad\qquad\quad = \sum_{e_{out} \in \mathit{outEdges}(\mathit{node})} [\mathit{timesTaken}(e_{out}) - \mathit{isStart}(e_{out})]\ ] \land \\
&\qquad ((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd}), \widehat{p}) \in \mathit{ImplicitDescrPath}
\end{aligned} \tag{A.61}$$

Expanding also the definition of *ImplicitDescrPath* (equation (5.83)), we obtain an equivalent statement with three additional equations exactly defining the values of the functions *timesTaken*, *isStart*, and *isEnd*.

$$
\begin{aligned}
&\forall \widehat{p} \in \widehat{Paths} : \\
&\quad \exists (timesTaken, isStart, isEnd) \in (Edges \to \mathbb{N}) \times (Edges \to \{0,1\})^2 : \\
&\qquad [\ \forall e \in Edges : isStart(e) \leq timesTaken(e)\ ] \wedge \\
&\qquad [\ \forall e \in Edges : isEnd(e) \leq timesTaken(e)\ ] \wedge \\
&\qquad \sum_{e \in Edges} isStart(e) \leq 1 \wedge \\
&\qquad \sum_{e \in Edges} isEnd(e) \leq 1 \wedge \\
&\qquad \sum_{e \in Edges} isStart(e) = \sum_{e \in Edges} isEnd(e) \wedge \\
&\qquad \sum_{e \in (Edges \setminus Edges_{start})} isStart(e) = 0 \wedge \\
&\qquad \sum_{e \in (Edges \setminus Edges_{end})} isEnd(e) = 0 \wedge \\
&\qquad [\ Nodes_{start} \cap Nodes_{end} = \emptyset \Rightarrow \sum_{e \in Edges} isStart(e) = 1\ ] \wedge \\
&\qquad [\ \forall node \in Nodes : \\
&\qquad\qquad \sum_{e_{in} \in inEdges(node)} [timesTaken(e_{in}) - isEnd(e_{in})] \\
&\qquad\qquad\quad = \sum_{e_{out} \in outEdges(node)} [timesTaken(e_{out}) - isStart(e_{out})]\ ] \wedge \\
&\qquad \forall e \in Edges : \\
&\qquad\quad timesTaken(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \wedge \\
&\qquad\quad isStart(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \wedge \\
&\qquad\quad isEnd(e) = \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right|
\end{aligned}
\tag{A.62}
$$

For each $\widehat{p} \in \widehat{Paths}$, there is one unique way to choose the three functions such that the three additional equations hold. As a consequence, we obtain an equivalent statement by removing the existential quantifier and the three equations and replacing all applications of the three functions by the corresponding right-hand sides of the removed equations.

$$
\begin{aligned}
&\forall \widehat{p} \in \widehat{Paths} : \\
&\quad [\ \forall e \in \mathit{Edges} : \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \\
&\qquad\qquad\qquad \leq \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right|\ ]\ \wedge \\
&\quad [\ \forall e \in \mathit{Edges} : \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \\
&\qquad\qquad\qquad \leq \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right|\ ]\ \wedge \\
&\quad [\ \sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \leq 1\ ]\ \wedge \\
&\quad [\ \sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \leq 1\ ]\ \wedge \\
&\quad [\ \sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \\
&\qquad = \sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right|\ ]\ \wedge \\
&\quad [\ \sum_{e \in (\mathit{Edges} \setminus \mathit{Edges}_{start})} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| = 0\ ]\ \wedge \qquad (A.63) \\
&\quad [\ \sum_{e \in (\mathit{Edges} \setminus \mathit{Edges}_{end})} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| = 0\ ]\ \wedge \\
&\quad [\ \mathit{Nodes}_{start} \cap \mathit{Nodes}_{end} = \emptyset \\
&\qquad \Rightarrow \sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| = 1\ ]\ \wedge \\
&\quad [\ \forall node \in \mathit{Nodes} : \\
&\qquad\qquad \sum_{e_{in} \in inEdges(node)} [\left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e_{in}\}\right| \\
&\qquad\qquad\qquad - \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e_{in}\}\right|] \\
&\qquad\quad = \sum_{e_{out} \in outEdges(node)} [\left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e_{out}\}\right| \\
&\qquad\qquad\qquad - \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e_{out}\}\right|]\ ]
\end{aligned}
$$

Thus, we end up at a conjunction over nine statements that have to hold for each $\widehat{p} \in \widehat{Paths}$. Now, we simply prove for each of these statements separately that it holds for each $\widehat{p} \in \widehat{Paths}$.

1. statement:

$$\begin{aligned}&[\ \forall e \in \mathit{Edges} : \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid x = 0 \land (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \\ &\qquad\qquad \leq \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right|\ ] \\ \Leftrightarrow\ &[\ \forall e \in \mathit{Edges} : 1\ ] \\ \Leftrightarrow\ &1\end{aligned}$$

2. statement:

$$\begin{aligned}&[\ \forall e \in \mathit{Edges} : \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid x = \mathit{len}(\widehat{p}) - 1 \land (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \\ &\qquad\qquad \leq \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right|\ ] \\ \Leftrightarrow\ &[\ \forall e \in \mathit{Edges} : 1\ ] \\ \Leftrightarrow\ &1\end{aligned}$$

3. statement:

$$\begin{aligned}&\sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid x = 0 \land (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \leq 1 \\ \Leftrightarrow\ &\min(1, \mathit{len}(\widehat{p})) \leq 1 \\ \Leftrightarrow\ &1\end{aligned}$$

4. statement:

$$\begin{aligned}&\sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid x = \mathit{len}(\widehat{p}) - 1 \land (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \leq 1 \\ \Leftrightarrow\ &\min(1, \mathit{len}(\widehat{p})) \leq 1 \\ \Leftrightarrow\ &1\end{aligned}$$

5. statement:

$$
\begin{aligned}
&\sum_{e\in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x),\widehat{p}(x+1)) = e\}\right| \\
&\quad = \sum_{e\in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x),\widehat{p}(x+1)) = e\}\right| \\
\Leftrightarrow &\begin{cases} 0 = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ 1 = 1 & , \mathit{else} \end{cases} \\
\Leftrightarrow &\, 1
\end{aligned}
$$

6. statement:

$$
\begin{aligned}
&\sum_{e\in(Edges\setminus Edges_{start})} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x),\widehat{p}(x+1)) = e\}\right| = 0 \\
\Leftrightarrow &\begin{cases} \displaystyle\sum_{e\in(Edges\setminus Edges_{start})} |\{\}| = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ \displaystyle\sum_{e\in(Edges\setminus Edges_{start})} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(0),\widehat{p}(1)) = e\}\right| = 0 & , \mathit{else} \end{cases} \\
\underset{\substack{(5.49)\\(5.56)}}{\Leftrightarrow} &\begin{cases} \displaystyle\sum_{e\in(Edges\setminus Edges_{start})} |\{\}| = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ \displaystyle\sum_{e\in(Edges\setminus Edges_{start})} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge \right. & , \mathit{else} \\ \qquad (\widehat{p}(0),\widehat{p}(1)) = e \wedge e \in Edges_{start}\}| = 0 & \end{cases} \\
\Leftrightarrow &\begin{cases} \displaystyle\sum_{e\in(Edges\setminus Edges_{start})} |\{\}| = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ \displaystyle\sum_{e\in(Edges\setminus Edges_{start})} |\{\}| = 0 & , \mathit{else} \end{cases} \\
\Leftrightarrow &\ 1
\end{aligned}
$$

7. statement:

$$
\begin{aligned}
& \sum_{e \in (Edges \setminus Edges_{end})} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| = 0 \\
\Leftrightarrow & \begin{cases} \sum_{e \in (Edges \setminus Edges_{end})} |\{\}| = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ \sum_{e \in (Edges \setminus Edges_{end})} \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge & , else \\ \quad (\widehat{p}(len(\widehat{p}) - 1), \widehat{p}(len(\widehat{p}))) = e\}| = 0 & \end{cases} \\
\underset{(5.57)}{\underset{(5.49)}{\Leftrightarrow}} & \begin{cases} \sum_{e \in (Edges \setminus Edges_{end})} |\{\}| = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ \sum_{e \in (Edges \setminus Edges_{end})} \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge & , else \\ \quad (\widehat{p}(len(\widehat{p}) - 1), \widehat{p}(len(\widehat{p}))) = e \wedge & \\ \quad e \in Edges_{end}\}| = 0 & \end{cases} \\
\Leftrightarrow & \begin{cases} \sum_{e \in (Edges \setminus Edges_{end})} |\{\}| = 0 & , \mathit{if}\ len(\widehat{p}) = 0 \\ \sum_{e \in (Edges \setminus Edges_{end})} |\{\}| = 0 & , else \end{cases} \\
\Leftrightarrow & \ 1
\end{aligned}
$$

8. statement:

$$
\begin{aligned}
& Nodes_{start} \cap Nodes_{end} = \emptyset \\
\Rightarrow & \ len(\widehat{p}) \geq 1 \\
\Rightarrow & \sum_{e \in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| = 1
\end{aligned}
$$

9. statement:

   For the case of $len(\widehat{p}) = 0$, the statement trivially holds as all sets in it coincide with the empty set. Thus, we only consider the case of $len(\widehat{p}) \neq 0$ here:

$$
\begin{aligned}
& \forall node \in Nodes : \\
& \qquad \sum_{e_{in} \in inEdges(node)} [\big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e_{in}\}\big| \\
& \qquad\qquad - \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e_{in}\}\big|] \\
& \qquad = \sum_{e_{out} \in outEdges(node)} [\big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e_{out}\}\big| \\
& \qquad\qquad - \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e_{out}\}\big|] \\
\Leftrightarrow\ & \forall node \in Nodes : \\
& \qquad \sum_{e_{in} \in inEdges(node)} \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e_{in}\}\big| \\
& \qquad - \sum_{e_{in} \in inEdges(node)} \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = len(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e_{in}\}\big| \\
& \qquad = \sum_{e_{out} \in outEdges(node)} \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e_{out}\}\big| \\
& \qquad - \sum_{e_{out} \in outEdges(node)} \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e_{out}\}\big| \\
\underset{(5.55)}{\underset{(5.54)}{\Leftrightarrow}}\ & \forall node \in Nodes : \\
& \qquad \big|\{x \in \mathbb{N}_{\leq len(\widehat{p})} \mid 0 < x \wedge \widehat{p}(x) = node\}\big| \\
& \qquad\qquad - \big|\{x \in \mathbb{N}_{\leq len(\widehat{p})} \mid x = len(\widehat{p}) \wedge \widehat{p}(x) = node\}\big| \\
& \qquad = \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid \widehat{p}(x) = node\}\big| \\
& \qquad\qquad - \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge \widehat{p}(x) = node\}\big| \\
\Leftrightarrow\ & \forall node \in Nodes : \\
& \qquad \big|\{x \in \mathbb{N}_{\leq len(\widehat{p})} \mid 0 < x \wedge \widehat{p}(x) = node\}\big| + \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \wedge \widehat{p}(x) = node\}\big| \\
& \qquad = \big|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid \widehat{p}(x) = node\}\big| + \big|\{x \in \mathbb{N}_{\leq len(\widehat{p})} \mid x = len(\widehat{p}) \wedge \widehat{p}(x) = node\}\big| \\
\Leftrightarrow\ & \forall node \in Nodes : \\
& \qquad \big|\{x \in \mathbb{N}_{\leq len(\widehat{p})} \mid \widehat{p}(x) = node\}\big| = \big|\{x \in \mathbb{N}_{\leq len(\widehat{p})} \mid \widehat{p}(x) = node\}\big| \\
\Leftrightarrow\ & 1 \quad \square
\end{aligned}
$$

Proof of Statement 5.95.

$$\begin{aligned}
&\sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w}(e) \\
\underset{\substack{(5.92)\\(5.83)\\(5.90)}}{=} &\begin{cases} \displaystyle\sum_{e \in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \cdot \widehat{w}(e) & , \mathit{if}\ \widehat{p} \in \widehat{Paths} \\ \displaystyle\sum_{e \in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid \widehat{p}(x) = e\}\right| \cdot \widehat{w}(e) & , \mathit{else} \end{cases} \\
= &\begin{cases} \displaystyle\sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{w}((\widehat{p}(x), \widehat{p}(x+1))) & , \mathit{if}\ \widehat{p} \in \widehat{Paths} \\ \displaystyle\sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{w}(\widehat{p}(x)) & , \mathit{else} \end{cases} \\
\underset{\substack{(5.61)\\(5.88)}}{=} &\sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{w}(\widehat{p}, x) \quad \square
\end{aligned}$$

Proof of Statement 5.96.

$$\begin{aligned}
&\sum_{e \in Edges} \mathit{isStart}(e) \cdot \widehat{w}(e) \\
\underset{\substack{(5.92)\\(5.83)\\(5.90)}}{=} &\begin{cases} \displaystyle\sum_{e \in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \land (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \cdot \widehat{w}(e) & , \mathit{if}\ \widehat{p} \in \widehat{Paths} \\ \displaystyle\sum_{e \in Edges} \left|\{x \in \mathbb{N}_{<len(\widehat{p})} \mid x = 0 \land \widehat{p}(x) = e\}\right| \cdot \widehat{w}(e) & , \mathit{else} \end{cases} \\
= &\begin{cases} \displaystyle\sum_{x \in (\mathbb{N}_{<len(\widehat{p})} \cap \{0\})} \widehat{w}((\widehat{p}(x), \widehat{p}(x+1))) & , \mathit{if}\ \widehat{p} \in \widehat{Paths} \\ \displaystyle\sum_{x \in (\mathbb{N}_{<len(\widehat{p})} \cap \{0\})} \widehat{w}(\widehat{p}(x)) & , \mathit{else} \end{cases} \\
\underset{\substack{(5.61)\\(5.88)}}{=} &\sum_{x \in (\mathbb{N}_{<len(\widehat{p})} \cap \{0\})} \widehat{w}(\widehat{p}, x) \quad \square
\end{aligned}$$

Proof of Statement 5.97.

$$
\begin{aligned}
&\sum_{e \in \mathit{Edges}} \mathit{isEnd}(e) \cdot \widehat{w}(e) \\
\underset{\substack{(5.92)\\(5.83)\\(5.90)}}{=}\ &\begin{cases}
\sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid x = \mathit{len}(\widehat{p}) - 1 \wedge (\widehat{p}(x), \widehat{p}(x+1)) = e\}\right| \cdot \widehat{w}(e) & \text{, } \mathit{if}\ \widehat{p} \in \widehat{\mathit{Paths}} \\
\sum_{e \in \mathit{Edges}} \left|\{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})} \mid x = \mathit{len}(\widehat{p}) - 1 \wedge \widehat{p}(x) = e\}\right| \cdot \widehat{w}(e) & \text{, } \mathit{else}
\end{cases} \\
=\ &\begin{cases}
\sum_{x \in (\mathbb{N}_{<\mathit{len}(\widehat{p})} \cap \{\mathit{len}(\widehat{p})-1\})} \widehat{w}((\widehat{p}(x), \widehat{p}(x+1))) & \text{, } \mathit{if}\ \widehat{p} \in \widehat{\mathit{Paths}} \\
\sum_{x \in (\mathbb{N}_{<\mathit{len}(\widehat{p})} \cap \{\mathit{len}(\widehat{p})-1\})} \widehat{w}(\widehat{p}(x)) & \text{, } \mathit{else}
\end{cases} \\
\underset{\substack{(5.61)\\(5.88)}}{=}\ &\sum_{x \in (\mathbb{N}_{<\mathit{len}(\widehat{p})} \cap \{\mathit{len}(\widehat{p})-1\})} \widehat{w}(\widehat{p}, x) \quad \square
\end{aligned}
$$

Soundness of the Lifting Rules in Table 5.4. We show for each lifting rule that it fulfills criterion (5.C5).

1. $\widehat{P^{path}}(\widehat{p}) \Leftrightarrow \forall z \in \mathbb{N}_{\leq len(\widehat{p})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_1}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_2}(\widehat{p}, x) + b_2$

$$
\begin{aligned}
& [\ \exists \widehat{p} \in \gamma_{impli}((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd})) : \widehat{P^{path}}(\widehat{p})\ ] \\
\Leftrightarrow\quad & [\ \exists \widehat{p} \in \gamma_{impli}((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd})) : \\
& \quad \forall z \in \mathbb{N}_{\leq len(\widehat{p})} : \\
& \qquad a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_1}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_2}(\widehat{p}, x) + b_2\ ] \\
\Rightarrow\quad & [\ \exists \widehat{p} \in \gamma_{impli}((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd})) : \\
& \quad \forall z \in (\mathbb{N}_{\leq len(\widehat{p})} \cap \{len(\widehat{p})\}) : \\
& \qquad a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_1}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_2}(\widehat{p}, x) + b_2\ ] \\
\Leftrightarrow\quad & [\ \exists \widehat{p} \in \gamma_{impli}((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd})) : \\
& \quad a_1 \cdot \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{w_1}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{w_2}(\widehat{p}, x) + b_2\ ] \\
\underset{(5.95)}{\Leftrightarrow}\quad & [\ \exists \widehat{p} \in \gamma_{impli}((\mathit{timesTaken}, \mathit{isStart}, \mathit{isEnd})) : \\
& \quad a_1 \cdot \sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w_1}(e) + b_1 \\
& \qquad \lesssim a_2 \cdot \sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w_2}(e) + b_2\ ] \\
\Rightarrow\quad & [\ a_1 \cdot \sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w_1}(e) + b_1 \\
& \quad \lesssim a_2 \cdot \sum_{e \in Edges} \mathit{timesTaken}(e) \cdot \widehat{w_2}(e) + b_2\ ] \\
\underset{(LR20)}{\Leftrightarrow}\quad & \boldsymbol{lift}\boldsymbol{(}\forall z \in \mathbb{N}_{\leq len(\widehat{p})} : a_1 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_1}(\widehat{p}, x) + b_1 \lesssim a_2 \cdot \sum_{x \in \mathbb{N}_{<z}} \widehat{w_2}(\widehat{p}, x) + b_2\boldsymbol{)} \\
\Leftrightarrow\quad & \boldsymbol{lift}\boldsymbol{(}\widehat{P^{path}}(\widehat{p})\boldsymbol{)} \quad \square
\end{aligned}
$$

Proof of Statement 6.41. Note that, during this proof, we do not explicitly write down the universal quantifiers for *prog* and C_i. Think of all the statements made during this proof to hold for all possible *prog* and C_i.

We mostly base this proof on the insights of Section 5.2. To this end, we start by relating $\mathit{ExecRuns}_{prog,C_i}$ and $\widehat{\mathit{ExecRuns}}_{prog,C_i}$ to further sets that are directly covered by the statements of Section 5.2.

$\mathit{ExecRuns}_{prog,C_i}$ is by definition (cf. equation (6.15)) a subset of the sequences of system states that start in an initial program state. We refer to this set as $\mathit{Traces}_{prog,C_i}$.

$$\mathit{ExecRuns}_{prog,C_i} \subseteq \mathit{Traces}_{prog,C_i} = \{t \in \mathit{Sequences} \mid t(0) \in \mathit{InitStates}_{prog,C_i}\} \tag{A.64}$$

Moreover, $\widehat{\mathit{ExecRuns}}_{prog,C_i}$ is by definition (cf. equation (6.36)) a subset of $\widehat{\mathit{Traces}}_{prog,C_i}$.

$$\widehat{\mathit{ExecRuns}}_{prog,C_i} \subseteq \widehat{\mathit{Traces}}_{prog,C_i} \tag{A.65}$$

We can interpret $\mathit{Traces}_{prog,C_i}$ as a set of concrete traces that has been defined analogously to Traces (cf. equation (5.4)). It only uses a different set of initial states ($\mathit{InitStates}_{prog,C_i}$ instead of S_{init}). In the same way, $\widehat{\mathit{Traces}}_{prog,C_i}$ has been defined analogously to $\widehat{\mathit{Traces}}$. Moreover, $\gamma_{trace,prog,C_i}$ is defined analogously to γ_{trace}. Thus, as a consequence of the insights of Section 5.2 and equation (6.34), $(\widehat{\mathit{Traces}}_{prog,C_i}, \gamma_{trace,prog,C_i})$ is an abstract model of $\mathit{Traces}_{prog,C_i}$.

$$\bigcup_{\hat{t} \in \widehat{\mathit{Traces}}_{prog,C_i}} \gamma_{trace,prog,C_i}(\hat{t}) \supseteq \mathit{Traces}_{prog,C_i} \tag{A.66}$$

Due to equation (A.64), it is also an abstract model of $\mathit{ExecRuns}_{prog,C_i}$.

$$\bigcup_{\hat{t} \in \widehat{\mathit{Traces}}_{prog,C_i}} \gamma_{trace,prog,C_i}(\hat{t}) \supseteq \mathit{ExecRuns}_{prog,C_i} \tag{A.67}$$

Every member of $\mathit{ExecRuns}_{prog,C_i}$ by definition (cf. equation (6.15)) fulfills the following helper property P_{help}.

$$P_{help}(t) \Leftrightarrow \forall x \in \mathbb{N}_{<len(t)-1} : \neg \mathit{End}_{prog,C_i}(t,x) \tag{A.68}$$

$$\forall t \in \mathit{ExecRuns}_{prog,C_i} : P_{help}(t) \tag{A.69}$$

Thus, we can easily lift P_{help} to $(\widehat{\mathit{Traces}}_{prog,C_i}, \gamma_{trace,prog,C_i})$ by applying the lifting rules of Section 5.2.

$$\begin{aligned}
& \widehat{P_{help}}(\hat{t}) \\
\Leftrightarrow\ & \boldsymbol{lift}(P_{help}(t)) \\
\underset{(A.68)}{\Leftrightarrow}\ & \boldsymbol{lift}(\forall x \in \mathbb{N}_{<len(t)-1} : \neg \mathit{End}_{prog,C_i}(t,x)) \\
\underset{\substack{(LR4)\\(5.32)}}{\Leftrightarrow}\ & \forall x \in \mathbb{N}_{<len(\hat{t})-1} : \boldsymbol{lift}(\neg \mathit{End}_{prog,C_i}(t,x)) \\
\underset{(LR6)}{\Leftrightarrow}\ & \forall x \in \mathbb{N}_{<len(\hat{t})-1} : \neg \boldsymbol{flip}(\boldsymbol{lift}(\mathit{End}_{prog,C_i}(t,x))) \\
\underset{\substack{(LR1)\\(LR9)}}{\Leftrightarrow}\ & \forall x \in \mathbb{N}_{<len(\hat{t})-1} : \neg \widehat{\mathit{End}}^{LB}_{prog,C_i}(\hat{t},x)
\end{aligned}$$

It follows that $\widehat{P_{help}}$ holds for each member of $\widehat{\mathit{ExecRuns}}_{prog,C_i}$ by definition.

$$\widehat{\mathit{ExecRuns}}_{prog,C_i} = \{\hat{t} \in \widehat{\mathit{Traces}}_{prog,C_i} \mid \widehat{P_{help}}(\hat{t})\} \tag{A.70}$$

Thus, statement (6.41) follows directly from the soundness of property lifting.

$$\bigcup_{\hat{t} \in \widehat{\mathit{ExecRuns}}_{prog,C_i}} \gamma_{trace,prog,C_i}(\hat{t}) \supseteq \mathit{ExecRuns}_{prog,C_i} \qquad \square$$

Proof Sketch for Statement 6.43. This proof is a slight extension of the proof of statement (6.41) that has been presented on page 277. Thus, we only present the needed addition in this proof sketch.

The essential idea is to not only lift the helper property P_{help} to $(\widehat{Traces_{prog,C_i}}, \gamma_{trace,prog,C_i})$, but also the $P_k \in Prop_{prog,C_i}$ since they are also guaranteed to hold for each member of $ExecRuns_{prog,C_i}$ (cf. equation (6.16)). In addition to $\widehat{P_{help}}$, we also use the lifted versions of these properties to prune infeasible members of $\widehat{Traces_{prog,C_i}}$. This results in the set $\widehat{LessExecRuns_{prog,C_i}}$.

$$\begin{aligned}
&\{\widehat{t} \in \widehat{Traces_{prog,C_i}} \mid \widehat{P_{help}}(\widehat{t}) \wedge \forall P_k \in Prop_{prog,C_i} : \widehat{P_k}(\widehat{t})\} \\
\underset{(A.70)}{=}\ &\{\widehat{t} \in \widehat{ExecRuns_{prog,C_i}} \mid \forall P_k \in Prop_{prog,C_i} : \widehat{P_k}(\widehat{t})\} \\
\underset{(6.42)}{=}\ &\widehat{LessExecRuns_{prog,C_i}}
\end{aligned}$$

Due to the soundness of property lifting, we follow that statement (6.43) holds.

$$\bigcup_{\widehat{t} \in \widehat{LessExecRuns_{prog,C_i}}} \gamma_{trace,prog,C_i}(\widehat{t}) \supseteq ExecRuns_{prog,C_i} \qquad \square$$

Proof of Statement 6.46. We begin this proof with the following auxiliary statement.

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores : \forall E \in Events : \\
&\quad \forall \widehat{t} \in \widehat{Traces_{prog,C_i}} : \forall t \in \gamma_{trace,prog,C_i}(\widehat{t}) : \\
&\qquad numEvOccur(prog, C_i, t, E) \leq \widehat{numEvOccur}(prog, C_i, \widehat{t}, E, UB)
\end{aligned} \tag{A.71}$$

It can be shown in the following way for all $\widehat{t} \in \widehat{Traces_{prog,C_i}}$ and all $t \in \gamma_{trace,prog,C_i}(\widehat{t})$.

$$\begin{aligned}
&numEvOccur(prog, C_i, t, E) \\
\underset{\substack{(6.20)\\(6.38)}}{=}\ &\left|\{x \in \mathbb{N}_{<len(t)} \mid Event_{prog,C_i,E}(t, x)\}\right| \\
\underset{(5.32)}{=}\ &\left|\{x \in \mathbb{N}_{<len(\widehat{t})} \mid Event_{prog,C_i,E}(t, x)\}\right| \\
\underset{(5.41)}{\leq}\ &\left|\{x \in \mathbb{N}_{<len(\widehat{t})} \mid \widehat{Event^{UB}_{prog,C_i,E}}(\widehat{t}, x)\}\right| \\
\underset{(6.44)}{=}\ &\widehat{numEvOccur}(prog, C_i, \widehat{t}, E, UB)
\end{aligned}$$

Subsequently, we can prove statement (6.46) as follows.

$$\begin{aligned}
&\widehat{Maximum_{prog,C_i,E}} \in \mathbb{N} \\
\underset{(6.45)}{\Rightarrow}\ &\forall \widehat{t} \in \widehat{LessExecRuns_{prog,C_i}} : \widehat{numEvOccur}(prog, C_i, \widehat{t}, E, UB) \leq \widehat{Maximum_{prog,C_i,E}} \\
\underset{\substack{(6.43)\\(A.71)}}{\Rightarrow}\ &\forall t \in ExecRuns_{prog,C_i} : numEvOccur(prog, C_i, t, E) \leq \widehat{Maximum_{prog,C_i,E}} \quad \square
\end{aligned}$$

Proof of Statement 6.52. In order to prove statement (6.52), we prove several auxiliary statements. The first auxiliary statement states that $\widehat{ExecRuns}^{term}_{prog,C_i}$ provides an overapproximation of $ExecRuns^{term}_{prog,C_i}$.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores :\\ &\quad \bigcup_{\widehat{t} \in \widehat{ExecRuns}^{term}_{prog,C_i}} \gamma_{trace,prog,C_i}(\widehat{t}) \supseteq ExecRuns^{term}_{prog,C_i}\end{aligned} \tag{A.72}$$

It is proved in the following way.

$$\begin{aligned}
& ExecRuns^{term}_{prog,C_i}\\
\underset{(6.23)}{=}\ & \{t \in ExecRuns_{prog,C_i} \mid len(t) \geq 1 \wedge End_{prog,C_i}(t, len(t) - 1)\}\\
\underset{\substack{(6.41)\\(5.32)\\(LR1)}}{\subseteq}\ & \bigcup\{\gamma_{trace,prog,C_i}(\widehat{t}) \mid \widehat{t} \in \widehat{ExecRuns}_{prog,C_i} \wedge len(\widehat{t}) \geq 1 \wedge \widehat{End}^{UB}_{prog,C_i}(\widehat{t}, len(\widehat{t}) - 1)\}\\
\underset{(6.47)}{=}\ & \bigcup_{\widehat{t} \in \widehat{ExecRuns}^{term}_{prog,C_i}} \gamma_{trace,prog,C_i}(\widehat{t})
\end{aligned}$$

The next auxiliary statement states that $\widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}$ provides an overapproximation of $ExecRuns^{diverg,end}_{prog,C_i,E}$.

$$\begin{aligned}&\forall prog \in Programs : \forall C_i \in Cores :\\ &\quad \bigcup_{\widehat{t} \in \widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}} \gamma_{trace,prog,C_i}(\widehat{t}) \supseteq ExecRuns^{diverg,end}_{prog,C_i,E}\end{aligned} \tag{A.73}$$

In order to prove it, we introduce an additional auxiliary statement.

$$\begin{aligned}
&\forall prog \in Programs : \forall C_i \in Cores :\\
&\quad \forall \widehat{t} \in \widehat{ExecRuns}_{prog,C_i} : \forall t, t' \in ExecRuns_{prog,C_i} :\\
&\quad\quad [\ t \in \gamma_{trace,prog,C_i}(\widehat{t}) \wedge (t, t') \in PrefixOf\,]\\
&\quad\quad \Rightarrow \exists \widehat{t'} \in \widehat{ExecRuns}_{prog,C_i} :\\
&\quad\quad\quad\quad t' \in \gamma_{trace,prog,C_i}(\widehat{t'}) \wedge (\widehat{t}, \widehat{t'}) \in \widehat{PrefixOf}
\end{aligned} \tag{A.74}$$

It is proved in the following way.

$$(\widehat{t},\widehat{u}) \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \wedge t,t' \in \mathit{ExecRuns}_{prog,C_i} \wedge t \in \gamma_{trace,prog,C_i}((\widehat{t},\widehat{u})) \wedge$$

$$(t,t') \in \mathit{PrefixOf}$$

$$\underset{(6.25)}{\Leftrightarrow} (\widehat{t},\widehat{u}) \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \wedge t,t' \in \mathit{ExecRuns}_{prog,C_i} \wedge t \in \gamma_{trace,prog,C_i}((\widehat{t},\widehat{u})) \wedge$$

$$\mathit{len}(t) \leq \mathit{len}(t') \wedge \forall x \in \mathbb{N}_{\leq \mathit{len}(t)} : t(x) = t'(x)$$

$$\underset{\substack{(5.3)\\(6.15)}}{\Leftrightarrow} (\widehat{t},\widehat{u}) \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \wedge t,t' \in \mathit{ExecRuns}_{prog,C_i} \wedge t \in \gamma_{trace,prog,C_i}((\widehat{t},\widehat{u})) \wedge$$

$$\mathit{len}(t) \leq \mathit{len}(t') \wedge [\forall x \in \mathbb{N}_{\leq \mathit{len}(t)} : t(x) = t'(x)] \wedge$$

$$[\forall x \in \mathbb{N}_{< \mathit{len}(t')} : \mathit{Cycle}(t',x)] \wedge$$

$$[\forall x \in \mathbb{N}_{< \mathit{len}(t')-1} : \neg \mathit{End}_{prog,C_i}(t',x)]$$

$$\underset{\substack{(6.40)\\(5.18)\\(5.17)}}{\Rightarrow} (\widehat{t},\widehat{u}) \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \wedge t,t' \in \mathit{ExecRuns}_{prog,C_i} \wedge$$

$$\mathit{len}(t) \leq \mathit{len}(t') \wedge [\forall x \in \mathbb{N}_{\leq \mathit{len}(t)} : t(x) = t'(x)] \wedge$$

$$[\forall x \in \mathbb{N}_{< \mathit{len}(t')} : \mathit{Cycle}(t',x)] \wedge$$

$$[\forall x \in \mathbb{N}_{< \mathit{len}(t')-1} : \neg \mathit{End}_{prog,C_i}(t',x)] \wedge$$

$$\mathit{len}((\widehat{t},\widehat{u})) = \mathit{len}(t) \wedge [\forall x \in \mathbb{N}_{\leq \mathit{len}(t)} : t(x) \in \gamma(\widehat{t}(x))] \wedge$$

$$t(\mathit{len}(t)) \in \gamma(\widehat{u}(\mathit{len}(t)))$$

$$\underset{\substack{(6.C2)\\(5.15)}}{\Rightarrow} (\widehat{t},\widehat{u}) \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \wedge t,t' \in \mathit{ExecRuns}_{prog,C_i} \wedge$$

$$[\forall x \in \mathbb{N}_{< \mathit{len}(t')-1} : \neg \mathit{End}_{prog,C_i}(t',x)] \wedge$$

$$[\exists (\widehat{t'},\widehat{u'}) \in \widehat{\mathit{Traces}}_{prog,C_i} :$$

$$t' \in \gamma_{trace,prog,C_i}((\widehat{t'},\widehat{u'})) \wedge \mathit{len}((\widehat{t},\widehat{u})) \leq \mathit{len}((\widehat{t'},\widehat{u'})) \wedge$$

$$\forall x \in \mathbb{N}_{\leq \mathit{len}((\widehat{t},\widehat{u}))} : \widehat{t'}(x) = \widehat{t}(x) \wedge \widehat{u'}(x) = \widehat{u}(x)]$$

$$\underset{(6.48)}{\Leftrightarrow} (\widehat{t},\widehat{u}) \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \wedge t,t' \in \mathit{ExecRuns}_{prog,C_i} \wedge$$

$$[\forall x \in \mathbb{N}_{< \mathit{len}(t')-1} : \neg \mathit{End}_{prog,C_i}(t',x)] \wedge$$

$$[\exists (\widehat{t'},\widehat{u'}) \in \widehat{\mathit{Traces}}_{prog,C_i} :$$

$$t' \in \gamma_{trace,prog,C_i}((\widehat{t'},\widehat{u'})) \wedge ((\widehat{t},\widehat{u}),(\widehat{t'},\widehat{u'})) \in \widehat{\mathit{PrefixOf}}]$$

$$\underset{(5.39)}{\Rightarrow} \exists(\widehat{t'}, \widehat{u'}) \in \widehat{Traces}_{prog,C_i} :$$

$$t' \in \gamma_{trace,prog,C_i}((\widehat{t'}, \widehat{u'})) \wedge ((\widehat{t}, \widehat{u}), (\widehat{t'}, \widehat{u'})) \in \widehat{PrefixOf} \wedge$$

$$\forall x \in \mathbb{N}_{<len(\widehat{t'})-1} : \neg \widehat{End}^{LB}_{prog,C_i}((\widehat{t'}, \widehat{u'}), x)$$

$$\underset{(6.36)}{\Leftrightarrow} \exists(\widehat{t'}, \widehat{u'}) \in \widehat{ExecRuns}_{prog,C_i} :$$

$$t' \in \gamma_{trace,prog,C_i}((\widehat{t'}, \widehat{u'})) \wedge ((\widehat{t}, \widehat{u}), (\widehat{t'}, \widehat{u'})) \in \widehat{PrefixOf}$$

We already know that $(\widehat{ExecRuns}_{prog,C_i}, \gamma_{trace,prog,C_i})$ is an abstract model of $ExecRuns^{diverg,end}_{prog,C_i,E}$.

$$\bigcup_{\widehat{t} \in \widehat{ExecRuns}_{prog,C_i}} \gamma_{trace,prog,C_i}(\widehat{t}) \underset{(6.41)}{\supseteq} ExecRuns_{prog,C_i} \underset{\substack{(6.28)\\(6.27)}}{\supseteq} ExecRuns^{diverg,end}_{prog,C_i,E}$$

Criterion (4.C1) is fulfilled as the restrictions in the definition of $ExecRuns^{diverg,end}_{prog,C_i,E}$ imply the restrictions in the definition of $\widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}$. We use auxiliary statement (A.74) in order to prove this. Let $\widehat{t} \in \widehat{ExecRuns}_{prog,C_i}$ in the following chain of implications.

$$\exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) :$$

$$t \in ExecRuns^{diverg,end}_{prog,C_i,E}$$

$$\underset{\substack{(6.28)\\(6.27)}}{\Rightarrow} \exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) :$$

$$t \in ExecRuns_{prog,C_i} \wedge$$

$$[len(t) = 0 \vee (len(t) > 0 \wedge Event_{prog,C_i,E}(t, len(t) - 1))] \wedge$$

$$[\forall n \in \mathbb{N} : \exists t' \in ExecRuns^{diverg}_{prog,C_i} :$$

$$(t, t') \in PrefixOf \wedge$$

$$len(t') = len(t) + n \wedge$$

$$\forall x \in \mathbb{N}_{\geq len(t)} \cap \mathbb{N}_{<len(t')} :$$

$$\neg Event_{prog,C_i,E}(t', x)]$$

$$\underset{\substack{(5.32)\\(LR1)}}{\Rightarrow} \exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) :$$

$$t \in ExecRuns_{prog,C_i} \ \wedge$$

$$[len(\widehat{t}) = 0 \vee (len(\widehat{t}) > 0 \wedge \widehat{Event}^{UB}_{prog,C_i,E}(\widehat{t}, len(\widehat{t}) - 1))] \wedge$$

$$[\forall n \in \mathbb{N} : \exists t' \in ExecRuns^{diverg}_{prog,C_i} :$$

$$(t, t') \in PrefixOf \wedge$$

$$len(t') = len(t) + n \wedge$$

$$\forall x \in \mathbb{N}_{\geq len(t)} \cap \mathbb{N}_{<len(t')} :$$

$$\neg Event_{prog,C_i,E}(t', x)]$$

$$\underset{(A.74)}{\Leftrightarrow} \exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) :$$

$$t \in ExecRuns_{prog,C_i} \ \wedge$$

$$[len(\widehat{t}) = 0 \vee (len(\widehat{t}) > 0 \wedge \widehat{Event}^{UB}_{prog,C_i,E}(\widehat{t}, len(\widehat{t}) - 1))] \wedge$$

$$[\forall n \in \mathbb{N} : \exists t' \in ExecRuns^{diverg}_{prog,C_i} :$$

$$(t, t') \in PrefixOf \wedge$$

$$[\exists \widehat{t'} \in \widehat{ExecRuns}_{prog,C_i} :$$

$$t' \in \gamma_{trace,prog,C_i}(\widehat{t'}) \wedge (\widehat{t}, \widehat{t'}) \in \widehat{PrefixOf}] \wedge$$

$$len(t') = len(t) + n \wedge$$

$$\forall x \in \mathbb{N}_{\geq len(t)} \cap \mathbb{N}_{<len(t')} :$$

$$\neg Event_{prog,C_i,E}(t', x)]$$

$$\underset{\substack{(5.32)\\(5.39)}}{\Rightarrow} [len(\widehat{t}) = 0 \vee (len(\widehat{t}) > 0 \wedge \widehat{Event}^{UB}_{prog,C_i,E}(\widehat{t}, len(\widehat{t}) - 1))] \wedge$$

$$[\forall n \in \mathbb{N} : \exists \widehat{t'} \in \widehat{ExecRuns}_{prog,C_i} :$$

$$(\widehat{t}, \widehat{t'}) \in \widehat{PrefixOf} \wedge$$

$$len(\widehat{t'}) = len(\widehat{t}) + n \wedge$$

$$\forall x \in \mathbb{N}_{\geq len(\widehat{t})} \cap \mathbb{N}_{<len(\widehat{t'})} :$$

$$\neg \widehat{Event}^{LB}_{prog,C_i,E}(\widehat{t'}, x)]$$

$$\underset{(6.50)}{\Leftrightarrow} \widehat{t} \in \widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}$$

As a consequence of the soundness of using properties lifted according to this criterion for pruning infeasible abstract traces from an abstract model (cf. equation (4.9)), auxiliary statement (A.73) holds.

Finally, statement (6.52) is a direct consequence of the auxiliary statements (A.72) and (A.73).

$$\begin{aligned}
&\bigcup_{\widehat{t} \in \widehat{ExecRuns}^{min\text{-}relev}_{prog,C_i,E}} \gamma_{trace,prog,C_i}(\widehat{t}) \\
\underset{(6.51)}{=}\; & \Big[\bigcup_{\widehat{t} \in \widehat{ExecRuns}^{term}_{prog,C_i}} \gamma_{trace,prog,C_i}(\widehat{t})\Big] \cup \Big[\bigcup_{\widehat{t} \in \widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}} \gamma_{trace,prog,C_i}(\widehat{t})\Big] \\
\underset{\substack{(A.72)\\(A.73)}}{\supseteq}\; & ExecRuns^{term}_{prog,C_i} \cup ExecRuns^{diverg,end}_{prog,C_i,E} \\
\underset{(6.29)}{=}\; & ExecRuns^{min\text{-}relev}_{prog,C_i,E} \quad \square
\end{aligned}$$

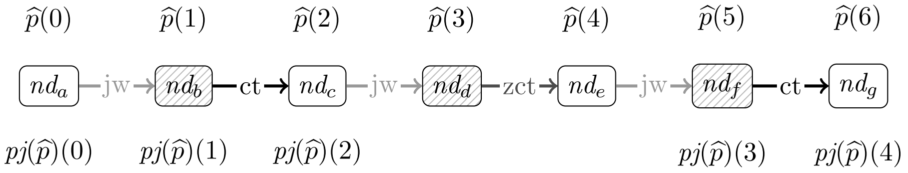
Proof Sketch for Statement 6.54. The formal proof of statement (6.54) can be performed by a slight extension to the proof of statement (6.52) as presented on page 280.

We know that the right-hand sides of the set inequations of auxiliary statements (A.72) and (A.73) are subsets of $ExecRuns_{prog,C_i}$. As a consequence of statement (6.16), each member of these subsets fulfills all the properties in $Prop_{prog,C_i}$.

We exploit this in two auxiliary statements that are almost identical to (A.72) and (A.73). The only difference is that—this time—they only argue about those members of $\widehat{ExecRuns}^{term}_{prog,C_i}$ respectively $\widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}$ that fulfill the lifted versions of all properties in $Prop_{prog,C_i}$. The proofs of these new auxiliary statements are performed analogously to the proofs of the original auxiliary statements. However, this time, we add the properties of $Prop_{prog,C_i}$ to the knowledge about the concrete traces. We lift them by the known lifting rules.

Finally, the union over the aforementioned subsets of $\widehat{ExecRuns}^{term}_{prog,C_i}$ and $\widehat{ExecRuns}^{diverg,end}_{prog,C_i,E}$ results in $Less\widehat{ExecRuns}_{prog,C_i}$. As a consequence of the new auxiliary statements, it overapproximates $ExecRuns_{prog,C_i}$. □

Proof of Statements 6.118 and 6.119. We begin this proof by introducing a projection function pj that operates on the members of $\widehat{RelPaths}^{detail,prog,C_i}$. It takes the sequence of nodes that makes up a relaxed path and removes all source and target nodes of zero cycle transition edges from it. We omit a formal definition of the function pj. Its principle is shown in the following figure. The white nodes in the figure belong to partition T, the shaded nodes to partition U. The edges are categorized as joining/widening (jw), cycle transition (ct), and zero cycle transition (zct) edges.



Note that the projected sequences of nodes might not correspond to a relaxed path of the graph. However, the projected sequences also still alternate between nodes of partition T and U. Furthermore, note that this projection might remove a number of nodes that is not a multiple of two in case the original relaxed path ends on a source node of a zero cycle transition edge.

Based on this, we define a mapping from relaxed paths of the detailed graph to sequences of abstract states. It is given by the function *map*, which is defined as follows.

$$\mathit{map} : \widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i} \rightarrow \widehat{\mathit{Sequences}} \tag{A.75}$$

$$\mathit{len}(\mathit{map}(\widehat{p})) = \left\lfloor \frac{\mathit{len}(\mathit{pj}(\widehat{p}))}{2} \right\rfloor \tag{A.76}$$

$$\mathit{map}(\widehat{p}) = (\widehat{t}, \widehat{u}), \text{ with } \begin{cases} \forall x \in \mathbb{N}_{\leq \frac{\mathit{len}(\mathit{pj}(\widehat{p}))}{2}} : \widehat{t}(x) = \mathit{toState}(\mathit{pj}(\widehat{p})(2 \cdot x)) \\ \forall x \in \mathbb{N}_{< \frac{\mathit{len}(\mathit{pj}(\widehat{p}))}{2}} : \widehat{u}(x) = \mathit{toState}(\mathit{pj}(\widehat{p})(2 \cdot x + 1)) \end{cases} \tag{A.77}$$

Note that due to the structural constraints of the detailed graph—which are similar to those on members of $\widehat{\mathit{Sequences}}$—it is always guaranteed that there is a member of $\widehat{\mathit{Sequences}}$ to which a relaxed path of the detailed graph can be mapped.

For relaxed paths $\widehat{p}$ with an odd $\mathit{len}(\mathit{pj}(\widehat{p}))$, there is only one possible choice for $\mathit{map}(\widehat{p}) = (\widehat{t}, \widehat{u}) \in \widehat{\mathit{Sequences}}$. For relaxed paths $\widehat{p}$ with an even $\mathit{len}(\mathit{pj}(\widehat{p}))$, however, the mapping rules we provided so far do not make a statement about the last position of $\widehat{u}$. Thus, only the general structural constraints for members of $\widehat{\mathit{Sequences}}$ are required. This can be observed for the relaxed path $\widehat{p}$ of the example figure above, which has an even $\mathit{len}(\widehat{p})$. It is mapped to a $(\widehat{t}, \widehat{u})$ in the following way.

$$\begin{aligned} \mathit{len}((\widehat{t}, \widehat{u})) &= 2 \\ \widehat{t}(0) &= \mathit{toState}(\mathit{nd}_a) \\ \widehat{u}(0) &= \mathit{toState}(\mathit{nd}_b) \\ \widehat{t}(1) &= \mathit{toState}(\mathit{nd}_c) \\ \widehat{u}(1) &= \mathit{toState}(\mathit{nd}_f) \\ \widehat{t}(2) &= \mathit{toState}(\mathit{nd}_g) \\ \widehat{u}(2) &\sqsupseteq \widehat{t}(2) \end{aligned}$$

In order to avoid conceptual problems, we additionally require the following from the mapping function *map*.

$$\begin{aligned} &\forall \widehat{p} \in \widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i} : \\ &\quad [\mathit{len}(\mathit{pj}(\widehat{p})) \equiv 0 \bmod 2 \wedge (\widehat{t}, \widehat{u}) = \mathit{map}(\widehat{p})] \Rightarrow \\ &\quad [\ [\exists \widehat{p}' \in \widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i} : \\ &\qquad\quad \mathit{len}(\mathit{pj}(\widehat{p}')) = \mathit{len}(\mathit{pj}(\widehat{p})) + 1 \wedge (\widehat{t}', \widehat{u}') = \mathit{map}(\widehat{p}') \wedge \\ &\qquad\quad \Big(\forall x \in \mathbb{N}_{\leq \mathit{len}((\widehat{t}, \widehat{u}))} : \widehat{t}'(x) = \widehat{t}(x)\Big) \wedge \\ &\qquad\quad \Big(\forall x \in \mathbb{N}_{< \mathit{len}((\widehat{t}, \widehat{u}))} : \widehat{u}'(x) = \widehat{u}(x)\Big)] \Rightarrow \\ &\qquad \exists \widehat{p}' \in \widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i} : \mathit{len}(\mathit{pj}(\widehat{p}')) = \mathit{len}(\mathit{pj}(\widehat{p})) + 1 \wedge \mathit{map}(\widehat{p}') = (\widehat{t}, \widehat{u})] \end{aligned} \tag{A.78}$$

Note that it is still guaranteed that there is a member of $\widehat{\mathit{Sequences}}$ to which a relaxed path of the detailed graph can be mapped. Moreover, note that it does not matter for our construction which choice the function *map* exactly makes. Any choice that fulfills the equations above is fine.

Now, we define the set $\widehat{\mathit{Mapped}}$ of all sequences of abstract states to which the members of $\widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i}$ are mapped.

$$\widehat{\mathit{Mapped}} = \{\mathit{map}(\widehat{p}) \mid \widehat{p} \in \widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i}\} \tag{A.79}$$

As a consequence of criterion (6.C3), the following statement holds.

$$\forall \widehat{s_i} \in \widehat{\mathit{InitStates}}_{prog,C_i} : \exists(\widehat{t},\widehat{u}) \in \widehat{\mathit{Mapped}} : \mathit{len}((\widehat{t},\widehat{u})) = 0 \land \widehat{t}(0) = \widehat{s_i} \tag{A.80}$$

Moreover, as a consequence of criteria (6.C4), (6.C5), and (6.C6) and equation (A.78), the following statement holds.

$$\begin{aligned}
&\forall(\widehat{t_1},\widehat{u_1}) \in \widehat{\mathit{Mapped}} :\\
&\quad [\forall x \in \mathbb{N}_{<\mathit{len}(\widehat{t})} : \neg \widehat{\mathit{End}}^{LB}_{prog,C_i}(\widehat{t},x)] \Rightarrow\\
&\quad \forall(\widehat{u_1}(\mathit{len}((\widehat{t_1},\widehat{u_1}))),\widehat{s_c}) \in \widehat{\mathit{Cycle}} :\\
&\qquad \exists(\widehat{t_2},\widehat{u_2}) \in \widehat{\mathit{Mapped}} :\\
&\qquad\quad \mathit{len}((\widehat{t_2},\widehat{u_2})) = 1 + \mathit{len}((\widehat{t_1},\widehat{u_1})) \ \land\\
&\qquad\quad (\forall x \in \mathbb{N}_{\leq \mathit{len}((\widehat{t_1},\widehat{u_1}))} : \widehat{t_2}(x) = \widehat{t_1}(x) \land \widehat{u_2}(x) = \widehat{u_1}(x)) \ \land\\
&\qquad\quad \widehat{t_2}(\mathit{len}((\widehat{t_2},\widehat{u_2}))) = \widehat{s_c}
\end{aligned} \tag{A.81}$$

Next, we chose a superset $\widehat{\mathit{Super}}$ of $\widehat{\mathit{Mapped}}$ in such a way that is fulfills statement (A.83), which is slightly stronger than statement (A.81).

$$\widehat{\mathit{Mapped}} \subseteq \widehat{\mathit{Super}} \subseteq \widehat{\mathit{Sequences}} \tag{A.82}$$

$$\begin{aligned}
&\forall(\widehat{t_1},\widehat{u_1}) \in \widehat{\mathit{Super}} :\\
&\quad \forall(\widehat{u_1}(\mathit{len}((\widehat{t_1},\widehat{u_1}))),\widehat{s_c}) \in \widehat{\mathit{Cycle}} :\\
&\qquad \exists(\widehat{t_2},\widehat{u_2}) \in \widehat{\mathit{Super}} :\\
&\qquad\quad \mathit{len}((\widehat{t_2},\widehat{u_2})) = 1 + \mathit{len}((\widehat{t_1},\widehat{u_1})) \ \land\\
&\qquad\quad (\forall x \in \mathbb{N}_{\leq \mathit{len}((\widehat{t_1},\widehat{u_1}))} : \widehat{t_2}(x) = \widehat{t_1}(x) \land \widehat{u_2}(x) = \widehat{u_1}(x)) \ \land\\
&\qquad\quad \widehat{t_2}(\mathit{len}((\widehat{t_2},\widehat{u_2}))) = \widehat{s_c}
\end{aligned} \tag{A.83}$$

Since $\widehat{\mathit{Super}}$ is a superset of $\widehat{\mathit{Mapped}}$, it automatically also fulfills statement (A.80).

$$\forall \widehat{s_i} \in \widehat{\mathit{InitStates}}_{prog,C_i} : \exists(\widehat{t},\widehat{u}) \in \widehat{\mathit{Super}} : \mathit{len}((\widehat{t},\widehat{u})) = 0 \land \widehat{t}(0) = \widehat{s_i} \tag{A.84}$$

As a consequence of equations (A.84) and (A.83), the set $\widehat{\mathit{Super}}$ fulfills all required criteria of $\widehat{\mathit{Traces}}_{prog,C_i}$, which are (6.C1) and (6.C2). Remember that $\widehat{\mathit{Traces}}_{prog,C_i}$ has not been defined in Section 6.3. It just has to be a subset of $\widehat{\mathit{Sequences}}$ which fulfills criteria (6.C1) and (6.C2). Thus, we simply define $\widehat{\mathit{Traces}}_{prog,C_i}$ to be identical to $\widehat{\mathit{Super}}$.

$$\widehat{\mathit{Traces}}_{prog,C_i} := \widehat{\mathit{Super}} \tag{A.85}$$

This implies that $\widehat{\mathit{ExecRuns}}_{prog,C_i}$ is a subset of $\widehat{\mathit{Mapped}}$.

$$\widehat{\mathit{ExecRuns}}_{prog,C_i} \underset{\substack{(A.85)\\(6.36)\\(A.81)}}{\subseteq} \widehat{\mathit{Mapped}} \tag{A.86}$$

It follows from equations (A.85), (A.82), and (6.60) and the choice of the edge weights in the detailed graph that every relaxed path of the detailed graph always describes the sequence of abstract states that it is mapped to by the function *map*.

$$\forall \widehat{p} \in \widehat{\mathit{RelPaths}}^{detail,prog,C_i} : \mathit{map}(\widehat{p}) \in \gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \tag{A.87}$$

Statement (6.118) is a direct consequence of this.

$$\bigcup_{\widehat{p}\in \widehat{\mathit{RelPaths}}^{detail,prog,C_i}} \gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \underset{\substack{(A.87)\\(A.79)}}{\supseteq} \widehat{\mathit{Mapped}} \underset{(A.86)}{\supseteq} \widehat{\mathit{ExecRuns}}_{prog,C_i}$$

Next, we define a subset of $\widehat{\mathit{Mapped}}$ that only contains those members which potentially end the program during their last cycle transition.

$$\widehat{\mathit{Mapped}}^{term} = \{\widehat{t} \in \widehat{\mathit{Mapped}} \mid \mathit{len}(\widehat{t}) \geq 1 \wedge \widehat{\mathit{End}}^{UB}_{prog,C_i}(\widehat{t}, \mathit{len}(\widehat{t}) - 1)\} \tag{A.88}$$

It follows that this subset is a superset of $\widehat{\mathit{ExecRuns}}^{term}_{prog,C_i}$.

$$\begin{aligned} &\widehat{\mathit{Mapped}}^{term} \\ \underset{(A.88)}{=}\ &\{\widehat{t} \in \widehat{\mathit{Mapped}} \mid \mathit{len}(\widehat{t}) \geq 1 \wedge \widehat{\mathit{End}}^{UB}_{prog,C_i}(\widehat{t}, \mathit{len}(\widehat{t}) - 1)\} \\ \underset{(A.86)}{\supseteq}\ &\{\widehat{t} \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \mid \mathit{len}(\widehat{t}) \geq 1 \wedge \widehat{\mathit{End}}^{UB}_{prog,C_i}(\widehat{t}, \mathit{len}(\widehat{t}) - 1)\} \\ \underset{(6.47)}{=}\ &\widehat{\mathit{ExecRuns}}^{term}_{prog,C_i} \end{aligned} \tag{A.89}$$

As a consequences of equations (6.114), (6.77), and (6.13), for every member of $\widehat{\mathit{Mapped}}^{term}$, there is a path through the detailed graph which is mapped to it.

$$\forall \widehat{t} \in \widehat{\mathit{Mapped}}^{term} : \exists \widehat{p} \in \widehat{\mathit{Paths}}^{detail,prog,C_i} : \widehat{t} = \mathit{map}(\widehat{p}) \tag{A.90}$$

Finally, statement (6.119) puts it all together.

$$\bigcup_{\widehat{p}\in \widehat{\mathit{Paths}}^{detail,prog,C_i}} \gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \underset{\substack{(A.87)\\(A.90)}}{\supseteq} \widehat{\mathit{Mapped}}^{term} \underset{(A.89)}{\supseteq} \widehat{\mathit{ExecRuns}}^{term}_{prog,C_i} \qquad \square$$

Proof of the Soundness of Using Properties Lifted to Sequences of Abstract States in Order to Safely Detect Infeasible Paths in a Detailed Graph. This proof reuses the machinery developed during the previous proof. The main idea is to only consider those paths $\widehat{p}$ for which all of the system properties lifted to sequences of abstract states hold for $\mathit{map}(\widehat{p})$. For the set of relaxed paths of the detailed graph, the soundness argument is provided as follows.

$$\begin{aligned} &\bigcup\{\gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \mid \widehat{p} \in \widehat{\mathit{RelPaths}}^{detail,prog,C_i} \wedge \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\mathit{map}(\widehat{p}))\} \\ \underset{\substack{(A.87)\\(A.79)}}{\supseteq}\ &\{\widehat{t} \in \widehat{\mathit{Mapped}} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\widehat{t})\} \\ \underset{(A.86)}{\supseteq}\ &\{\widehat{t} \in \widehat{\mathit{ExecRuns}}_{prog,C_i} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\widehat{t})\} \end{aligned}$$

For the paths through the detailed graph, the soundness argument is provided as follows.

$$\begin{aligned} &\bigcup\{\gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \mid \widehat{p} \in \widehat{\mathit{Paths}}^{detail,prog,C_i} \wedge \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\mathit{map}(\widehat{p}))\} \\ \underset{\substack{(A.87)\\(A.90)}}{\supseteq}\ &\{\widehat{t} \in \widehat{\mathit{Mapped}}^{term} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\widehat{t})\} \\ \underset{(A.89)}{\supseteq}\ &\{\widehat{t} \in \widehat{\mathit{ExecRuns}}^{term}_{prog,C_i} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\widehat{t})\} \end{aligned}$$

Note that—in an actual implementation—one does not really have to implement a function *map*. The same effect is achieved by altering the properties on sequences of abstract states in the following way:

- Replace $len(\widehat{t})$ by the number of cycle transition edges passed along $\widehat{p}$ increased by one.
- Instead of $\widehat{E^{UB}}(\widehat{t}, x)$, use the upper-bounding edge weight for E of the x-th cycle transition edge along $\widehat{p}$.
- Instead of $\widehat{E^{LB}}(\widehat{t}, x)$, use the lower-bounding edge weight for E of the x-th cycle transition edge along $\widehat{p}$. □

Proof of Statement 6.121. Formally, we proceed by proving the following statement, which directly implies statement (6.121).

$$\begin{aligned}&\forall \widehat{t} \in \widehat{Traces}_{prog,C_i} \cup \widehat{SpuriousTraces}_{prog,C_i} :\\ &\quad [\widehat{p_B} \vDash_{path} \widehat{p_A} \land \widehat{t} \in \gamma^{A}_{path,prog,C_i}(\widehat{p_A})] \Rightarrow \widehat{t} \in \gamma^{B}_{path,prog,C_i}(\widehat{p_B})\end{aligned} \tag{A.91}$$

First, we take a look at an implication of $\widehat{t} \in \gamma^{A}_{path,prog,C_i}(\widehat{p_A})$.

$$\begin{aligned}&\widehat{t} \in \gamma^{A}_{path,prog,C_i}(\widehat{p_A})\\ \underset{\substack{(5.78)\\(5.69)\\(5.76)}}{\Rightarrow} &\exists part_{inner} \in Partitionings(len(\widehat{p_A}), len(\widehat{t})) :\\ &\quad \forall E \in Events :\\ &\quad\quad \forall x \in \mathbb{N}_{<len(\widehat{p_A})} :\\ &\quad\quad\quad \sum_{from(part_{inner},x) \leq i \leq to(part_{inner},x)} \widehat{E^{UB}}(\widehat{t}, i) \leq \widehat{wE^{UB}}(\widehat{p_A}, x) \land\\ &\quad\quad\quad \sum_{from(part_{inner},x) \leq i \leq to(part_{inner},x)} \widehat{E^{LB}}(\widehat{t}, i) \geq \widehat{wE^{LB}}(\widehat{p_A}, x)\end{aligned} \tag{A.92}$$

Then, we take a look at an implication of $\widehat{p_B} \vDash_{path} \widehat{p_A}$.

$$\begin{aligned}&\widehat{p_B} \vDash_{path} \widehat{p_A}\\ \underset{(6.120)}{\Rightarrow} &\exists part_{outer} \in Partitionings(len(\widehat{p_B}), len(\widehat{p_A})) :\\ &\quad \forall E \in Events :\\ &\quad\quad \forall x \in \mathbb{N}_{<len(\widehat{p_B})} :\\ &\quad\quad\quad \sum_{from(part_{outer},x) \leq i \leq to(part_{outer},x)} \widehat{wE^{UB}}(\widehat{p_A}, i) \leq \widehat{wE^{UB}}(\widehat{p_B}, x) \land\\ &\quad\quad\quad \sum_{from(part_{outer},x) \leq i \leq to(part_{outer},x)} \widehat{wE^{LB}}(\widehat{p_A}, i) \geq \widehat{wE^{LB}}(\widehat{p_B}, x)\end{aligned} \tag{A.93}$$

Next, we combine the existing $part_{inner}$ and $part_{outer}$ to a $part_{comb}$ in the following way.

$$part_{comb} \in Partitionings(len(\widehat{p_B}), len(\widehat{t})) \tag{A.94}$$

$$\begin{aligned}&\forall x \in \mathbb{N}_{<len(\widehat{p_B})} :\\ &\quad part_{comb}(x) = \sum_{from(part_{outer},x) \leq i \leq to(part_{outer},x)} part_{inner}(i)\end{aligned} \tag{A.95}$$

From these equations, we follow another statement that directly implies what we initially wanted to show.

$$
\begin{aligned}
&\underset{\substack{(A.92)\\(A.93)\\(A.95)}}{\Rightarrow} \forall E \in \mathit{Events}:\\
&\qquad \forall x \in \mathbb{N}_{<len(\widehat{p_B})}:\\
&\qquad\quad \sum_{from(part_{comb},x)\leq i\leq to(part_{comb},x)} \widehat{E^{UB}}(\widehat{t},i) \leq \widehat{wE^{UB}}(\widehat{p_B},x) \wedge\\
&\qquad\quad \sum_{from(part_{comb},x)\leq i\leq to(part_{comb},x)} \widehat{E^{LB}}(\widehat{t},i) \geq \widehat{wE^{LB}}(\widehat{p_B},x)\\
&\underset{\substack{(5.78)\\(5.69)\\(5.76)\\(A.94)}}{\Rightarrow} \widehat{t} \in \gamma^{B}_{path,prog,C_i}(\widehat{p_B})
\end{aligned}
\qquad \square
$$

Proof of Statements 6.129 and 6.130. In this proof, we show the following two statements, which directly imply statements (6.129) and (6.130).

$$
\begin{aligned}
&\forall E \in \mathit{Events}:\\
&\quad G^{B} \models G^{detail,prog,C_i} \Rightarrow\\
&\quad \forall t \in \mathit{ExecRuns}_{prog,C_i} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^{B}_{prog,C_i} :\\
&\qquad \mathit{numEvOccur}(prog,C_i,t,E) \leq \sum_{x\in\mathbb{N}_{<len(\widehat{p})}} \widehat{\mathit{wEvent}}^{UB}_{prog,C_i,E}(\widehat{p},x)
\end{aligned}
\tag{A.96}
$$

$$
\begin{aligned}
&\forall E \in \mathit{Events}:\\
&\quad G^{B} \models G^{detail,prog,C_i} \Rightarrow\\
&\quad \forall t \in \mathit{ExecRuns}^{min\text{-}relev}_{prog,C_i,E} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^{B}_{prog,C_i} :\\
&\qquad \mathit{numEvOccur}(prog,C_i,t,E) \geq \sum_{x\in\mathbb{N}_{<len(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p},x)
\end{aligned}
\tag{A.97}
$$

We start this proof by considering the terminated execution runs. They are overapproximated by the set $\widehat{\mathit{ExecRuns}}^{term}_{prog,C_i}$.

$$
\begin{aligned}
&\mathit{ExecRuns}^{term}_{prog,C_i}\\
\underset{\substack{(6.41)\\(5.32)\\(LR1)}}{\subseteq}\; &\bigcup\{\gamma_{trace,prog,C_i}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{ExecRuns}}^{term}_{prog,C_i}\}
\end{aligned}
\tag{A.98}
$$

$\widehat{\mathit{ExecRuns}}^{term}_{prog,C_i}$ is again overapproximated by the paths through graph G^{B} which subsumes the detailed graph.

$$
\begin{aligned}
&\widehat{\mathit{ExecRuns}}^{term}_{prog,C_i}\\
\underset{(6.119)}{\subseteq}\; &\bigcup\{\gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \mid \widehat{p} \in \widehat{\mathit{Paths}^{detail,prog,C_i}}\}\\
\underset{\substack{(6.C7)\\(6.121)}}{\subseteq}\; &\bigcup\{\gamma^{B}_{path,prog,C_i}(\widehat{p}) \mid \widehat{p} \in \widehat{\mathit{Paths}^{B}}\}
\end{aligned}
\tag{A.99}
$$

Due to the soundness of property lifting, we can enrich both statements with lifted properties.

$$\begin{aligned}&\mathit{ExecRuns}^{term}_{prog,C_i}\\ \underset{(A.98)}{\subseteq}\ &\bigcup\{\gamma_{trace,prog,C_i}(\widehat{t}) \mid \widehat{t} \in \widehat{\mathit{ExecRuns}}^{term}_{prog,C_i} \wedge \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\widehat{t})\}\end{aligned} \tag{A.100}$$

$$\begin{aligned}&\{\widehat{t} \in \widehat{\mathit{ExecRuns}}^{term}_{prog,C_i} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P_k}(\widehat{t})\}\\ \underset{(A.99)}{\subseteq}\ &\bigcup\{\gamma^B_{path,prog,C_i}(\widehat{p}) \mid \widehat{p} \in \widehat{\mathit{Paths}^B} \wedge \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P^{path}_k}(\widehat{p})\}\end{aligned} \tag{A.101}$$

Note that the paths through graph G^B that fulfill the lifted versions of all system properties are a subset of $\widehat{\mathit{LessFeedPaths}}^B_{prog,C_i}$.

$$\{\widehat{p} \in \widehat{\mathit{Paths}^B} \mid \forall P_k \in \mathit{Prop}_{prog,C_i} : \widehat{P^{path}_k}(\widehat{p})\} \subseteq \widehat{\mathit{LessFeedPaths}}^B_{prog,C_i} \tag{A.102}$$

Moreover, we know that the number of event occurrences in each of the prefixes of a terminated execution run is upper-bounded by the number of event occurrences in one of the terminated execution runs.

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad \forall t' \in \bigcup_{t \in \mathit{ExecRuns}^{term}_{prog,C_i}} \{t' \mid (t',t) \in \mathit{PrefixOf}\} : \exists t \in \mathit{ExecRuns}^{term}_{prog,C_i} :\\ &\qquad \mathit{numEvOccur}(prog, C_i, t', E) \leq \mathit{numEvOccur}(prog, C_i, t, E)\end{aligned} \tag{A.103}$$

Statements (A.100), (A.101), (5.42), (5.69), (A.102), and (A.103) imply that the following two statements hold.

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad G^B \models G^{detail,prog,C_i} \Rightarrow\\ &\quad \forall t \in \bigcup_{t' \in \mathit{ExecRuns}^{term}_{prog,C_i}} \{t \mid (t,t') \in \mathit{PrefixOf}\} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^B_{prog,C_i} :\\ &\qquad \mathit{numEvOccur}(prog, C_i, t, E) \leq \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{\mathit{wEvent}}^{UB}_{prog,C_i,E}(\widehat{p}, x)\end{aligned} \tag{A.104}$$

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad G^B \models G^{detail,prog,C_i} \Rightarrow\\ &\quad \forall t \in \mathit{ExecRuns}^{term}_{prog,C_i} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^B_{prog,C_i} :\\ &\qquad \mathit{numEvOccur}(prog, C_i, t, E) \geq \sum_{x \in \mathbb{N}_{<len(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p}, x)\end{aligned} \tag{A.105}$$

Thus, we already showed a significant part of statements (A.96) and (A.97). In order to fully prove these statements, according to equations (6.27) and (6.29), we are only left to show the following two statements.

$$\begin{aligned}
&\forall E \in \mathit{Events} : \\
&\quad G^{B} \models G^{\mathit{detail},\mathit{prog},C_i} \Rightarrow \\
&\quad \forall t \in \mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^{B}_{\mathit{prog},C_i} : \\
&\qquad \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) \leq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{\mathit{UB}}_{\mathit{prog},C_i,E}(\widehat{p}, x)
\end{aligned} \tag{A.106}$$

$$\begin{aligned}
&\forall E \in \mathit{Events} : \\
&\quad G^{B} \models G^{\mathit{detail},\mathit{prog},C_i} \Rightarrow \\
&\quad \forall t \in \mathit{ExecRuns}^{\mathit{diverg},\mathit{end}}_{\mathit{prog},C_i,E} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^{B}_{\mathit{prog},C_i} : \\
&\qquad \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) \geq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{\mathit{LB}}_{\mathit{prog},C_i,E}(\widehat{p}, x)
\end{aligned} \tag{A.107}$$

This means that the remainder of this proof only has to consider the set $\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}$ of diverging execution run prefixes as the two statements only argue about concrete traces contained in this set. As a consequence, the remainder of this proof only considers abstract traces of the respective abstract models that are feasible with respect to $\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}$. A sequence of abstract states in $\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i}$ is feasible with respect to $\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}$ if and only if it describes at least one prefix of a member of $\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}$.

$$\mathit{prefixes}(\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}) = \{t \mid \exists t' \in \mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i} : (t, t') \in \mathit{PrefixOf}\} \tag{A.108}$$

$$\begin{aligned}
\mathit{feas}(\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i}) = \{&\widehat{t} \in \widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i} \mid \\
&\gamma_{\mathit{trace},\mathit{prog},C_i}(\widehat{t}) \cap \mathit{prefixes}(\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}) \neq \emptyset\}
\end{aligned} \tag{A.109}$$

It follows from equation (6.41) that these feasible sequences of abstract states provide an overapproximation of $\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}$.

$$\begin{aligned}
&\forall \mathit{prog} \in \mathit{Programs} : \forall C_i \in \mathit{Cores} : \\
&\quad \bigcup_{\widehat{t} \in \mathit{feas}(\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i})} \gamma_{\mathit{trace},\mathit{prog},C_i}(\widehat{t}) \supseteq \mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}
\end{aligned} \tag{A.110}$$

Analogously, a relaxed path of the detailed graph is feasible with respect to $\mathit{ExecRuns}^{\mathit{diverg}}_{\mathit{prog},C_i}$ if and only if it is mapped to a member of $\mathit{feas}(\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i})$. The mapping function *map* is borrowed from the proof of statements (6.118) and (6.119) as found on page 284.

$$\begin{aligned}
\mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i}) = \{&\widehat{p} \in \widehat{\mathit{RelPaths}}^{\mathit{detail},\mathit{prog},C_i} \mid \\
&\mathit{map}(\widehat{p}) \in \mathit{feas}(\widehat{\mathit{ExecRuns}}_{\mathit{prog},C_i})\}
\end{aligned} \tag{A.111}$$

It follows that these feasible relaxed paths of the detailed graph provide an overapproximation of $\mathit{feas}(\widehat{\mathit{ExecRuns}_{prog,C_i}})$.

$$\begin{aligned} & \bigcup_{\widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}})} \gamma^{detail,prog,C_i}_{path,prog,C_i}(\widehat{p}) \\ \underset{(A.87)}{\supseteq} & \bigcup_{\widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}})} \mathit{map}(\widehat{p}) \\ \underset{(A.86)}{\supseteq} & \mathit{feas}(\widehat{\mathit{ExecRuns}_{prog,C_i}}) \end{aligned} \tag{A.112}$$

Next, we present three small lemmas about the feasible relaxed paths of the detailed graph. We do not provide formal proofs of these lemmas as these proofs are tedious and lengthy. Thus, we resort to a short intuition per lemma.

The first lemma states that every prefix of a feasible relaxed path of the detailed graph is also feasible. Intuitively, if a relaxed path is mapped to a sequence abstract states that describes members of $\mathit{ExecRuns}^{diverg}_{prog,C_i}$, any prefix of the relaxed path is mapped to a (potentially shorter) sequence that also describes members of $\mathit{ExecRuns}^{diverg}_{prog,C_i}$.

$$\begin{aligned} & \forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \forall \widehat{p}' \in \widehat{\mathit{RelPaths}^{detail,prog,C_i}} : \\ & \quad (\widehat{p}', \widehat{p}) \in \widehat{\mathit{PrefixOf}^{detail,prog,C_i}_{path}} \Rightarrow \\ & \quad \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) \end{aligned} \tag{A.113}$$

The second lemma states that every feasible relaxed path of the detailed graph has a strict extension that is also feasible. Intuitively, any member of $\mathit{ExecRuns}^{diverg}_{prog,C_i}$ can be extended arbitrary long without reaching the program end (cf. equations (6.24) and (6.27)). Thus, any feasible relaxed path of the detailed graph can also be extended arbitrary long without becoming infeasible with respect to $\mathit{ExecRuns}^{diverg}_{prog,C_i}$.

$$\begin{aligned} & \forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \\ & \quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}^{detail,prog,C_i}_{path}} \wedge \mathit{len}(\widehat{p}') = \mathit{len}(\widehat{p}) + 1 \end{aligned} \tag{A.114}$$

The third lemma states that a relaxed path of the detailed graph has arbitrary long feasible extensions that do not guarantee further occurrences of event E if it is mapped to a sequence of abstract states that describes a member of $\mathit{ExecRuns}^{diverg,end}_{prog,C_i,E}$. Intuitively, any member of $\mathit{ExecRuns}^{diverg,end}_{prog,C_i,E}$ can be extended arbitrary long without further occurrences of event E (cf. equation (6.28)). Thus, the same can be observed for the feasible relaxed paths of the detailed graph that are mapped to a sequence which describes a member of $\mathit{ExecRuns}^{diverg,end}_{prog,C_i,E}$.

$$\begin{aligned} & \forall E \in \mathit{Events} : \\ & \quad \forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \\ & \quad\quad [\exists t \in \mathit{ExecRuns}^{diverg,end}_{prog,C_i,E} : t \in \gamma_{trace,prog,C_i}(\mathit{map}(\widehat{p}))] \Rightarrow \\ & \quad\quad \forall n \in \mathbb{N} : \exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \\ & \quad\quad\quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}^{detail,prog,C_i}_{path}} \wedge \\ & \quad\quad\quad \mathit{len}(\widehat{p}') = \mathit{len}(\widehat{p}) + n \wedge \\ & \quad\quad\quad \forall x \in \mathbb{N}_{\geq \mathit{len}(\widehat{p})} \cap \mathbb{N}_{< \mathit{len}(\widehat{p}')} : \widehat{\mathit{wEvent}^{LB}_{prog,C_i,E}}(\widehat{p}', x) = 0 \end{aligned} \tag{A.115}$$

Next, we take a closer look at graph G^B which subsumes the detailed graph. A relaxed path of G^B is feasible with respect to $\mathit{ExecRuns}^{diverg}_{prog,C_i}$ if and only if it describes members of $\mathit{feas}(\widehat{\mathit{ExecRuns}}_{prog,C_i})$.

$$\mathit{feas}(\widehat{\mathit{RelPaths}^B}) = \{\widehat{p} \in \widehat{\mathit{RelPaths}^B} \mid \gamma^B_{path,prog,C_i}(\widehat{p}) \cap \mathit{feas}(\widehat{\mathit{ExecRuns}}_{prog,C_i}) \neq \emptyset\} \tag{A.116}$$

In the following, we present the four key statements of this proof. They create a connection between the feasible relaxed paths of G^B and those of the detailed graph. The subsumption relation between G^B and the detailed graph is sufficient to prove the four key statements. However, we do not present full formal proofs of the four key statements. Instead, we briefly sketch a sufficient induction hypothesis per key statement.

The first key statement points out that, for every feasible relaxed path of the detailed graph, there is a feasible extension which is represented by a feasible relaxed path of G^B.

$$\begin{aligned}
&\forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \\
&\quad \exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) : \\
&\quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}^{detail,prog,C_i}_{path}} \wedge \\
&\quad \exists \widehat{p_B} \in \mathit{feas}(\widehat{\mathit{RelPaths}^B}) : \widehat{p_B} \vdash_{path} \widehat{p}'
\end{aligned} \tag{A.117}$$

The proof of this key statement argues about the set $\mathit{PrefExt}(\widehat{p})$ of all feasible prefixes and extensions of a given feasible relaxed path $\widehat{p}$ of the detailed graph.

$$\mathit{PrefExt}(\widehat{p}) = \{\widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}^{detail,prog,C_i}}) \mid \{(\widehat{p}', \widehat{p}), (\widehat{p}, \widehat{p}')\} \cap \widehat{\mathit{PrefixOf}^{detail,prog,C_i}_{path}} \neq \emptyset\}$$

In the inductive part of the proof, we prove the following hypothesis IH_1 for all $i \in \mathbb{N}$. The first key statement follows from $\mathit{IH}_1(\mathit{len}(\widehat{p}))$.

$$\begin{aligned}
&\mathit{IH}_1(i) \\
\Leftrightarrow &\exists \widehat{p}' \in \mathit{PrefExt}(\widehat{p}) : \\
&\quad \mathit{len}(\widehat{p}') \geq i \wedge \exists \widehat{p_B} \in \widehat{\mathit{RelPaths}^B} : \widehat{p_B} \vdash_{path} \widehat{p}'
\end{aligned}$$

The second key statement is a slight variant of the first key statement. It states that, for every feasible relaxed path $\widehat{p}$ of the detailed graph which has arbitrary long feasible extensions that do not guarantee further occurrences of event E, there is a feasible extension of $\widehat{p}$ which is represented by a feasible relaxed path $\widehat{p_B}$ of G^B and which has arbitrary long feasible extensions that do not guarantee more occurrences of event E than $\widehat{p}$ does.

$$\begin{aligned}
&\forall E \in \mathit{Events}: \\
&\quad \forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail,prog},C_i}): \\
&\qquad [\forall n \in \mathbb{N} : \exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail,prog},C_i}): \\
&\qquad\quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}}_{\mathit{path}}^{\mathit{detail,prog},C_i} \ \wedge \\
&\qquad\quad \mathit{len}(\widehat{p}') = \mathit{len}(\widehat{p}) + n \ \wedge \\
&\qquad\quad \forall x \in \mathbb{N}_{\geq \mathit{len}(\widehat{p})} \cap \mathbb{N}_{< \mathit{len}(\widehat{p}')} : \widehat{\mathit{wEvent}}^{LB}_{\mathit{prog},C_i,E}(\widehat{p}', x) = 0] \ \Rightarrow \\
&\qquad \exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail,prog},C_i}): \\
&\qquad\quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}}_{\mathit{path}}^{\mathit{detail,prog},C_i} \ \wedge \\
&\qquad\quad [\exists \widehat{p_B} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) : \widehat{p_B} \vdash_{\mathit{path}} \widehat{p}'] \ \wedge \\
&\qquad\quad \forall n \in \mathbb{N} : \exists \widehat{p}'' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail,prog},C_i}): \\
&\qquad\qquad (\widehat{p}', \widehat{p}'') \in \widehat{\mathit{PrefixOf}}_{\mathit{path}}^{\mathit{detail,prog},C_i} \ \wedge \\
&\qquad\qquad \mathit{len}(\widehat{p}'') = \mathit{len}(\widehat{p}') + n \ \wedge \\
&\qquad\qquad \forall x \in \mathbb{N}_{\geq \mathit{len}(\widehat{p})} \cap \mathbb{N}_{< \mathit{len}(\widehat{p}'')} : \widehat{\mathit{wEvent}}^{LB}_{\mathit{prog},C_i,E}(\widehat{p}'', x) = 0
\end{aligned} \tag{A.118}$$

The proof of the second key statement is very similar to that of the first key statement. It argues about the set $\mathit{PrefExt}(\widehat{p}, E)$. It contains all feasible prefixes of a given feasible relaxed path $\widehat{p}$ of the detailed graph. Moreover, it contains those feasible extensions of $\widehat{p}$ which again have arbitrary long feasible extensions that do not guarantee more occurrences of event E than $\widehat{p}$ does.

$$\begin{aligned}
\mathit{PrefExt}(\widehat{p}, E) = \{\widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail,prog},C_i}) \mid{} & (\widehat{p}', \widehat{p}) \in \widehat{\mathit{PrefixOf}}_{\mathit{path}}^{\mathit{detail,prog},C_i} \ \vee \\
& [(\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}}_{\mathit{path}}^{\mathit{detail,prog},C_i} \ \wedge \\
& \ \forall n \in \mathbb{N} : \exists \widehat{p}'' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{\mathit{detail,prog},C_i}): \\
& \quad (\widehat{p}', \widehat{p}'') \in \widehat{\mathit{PrefixOf}}_{\mathit{path}}^{\mathit{detail,prog},C_i} \ \wedge \\
& \quad \mathit{len}(\widehat{p}'') = \mathit{len}(\widehat{p}') + n \ \wedge \\
& \quad \forall x \in \mathbb{N}_{\geq \mathit{len}(\widehat{p})} \cap \mathbb{N}_{< \mathit{len}(\widehat{p}'')} : \widehat{\mathit{wEvent}}^{LB}_{\mathit{prog},C_i,E}(\widehat{p}'', x) = 0]\}
\end{aligned}$$

In the inductive part of the proof, we prove the following hypothesis IH_2 for all $i \in \mathbb{N}$. The second key statement follows from $\mathit{IH}_2(\mathit{len}(\widehat{p}))$.

$$\begin{aligned}
&\mathit{IH}_2(i) \\
\Leftrightarrow & \exists \widehat{p}' \in \mathit{PrefExt}(\widehat{p}, E): \\
&\quad \mathit{len}(\widehat{p}') \geq i \wedge \exists \widehat{p_B} \in \widehat{\mathit{RelPaths}}^{B} : \widehat{p_B} \vdash_{\mathit{path}} \widehat{p}'
\end{aligned}$$

The third key statement points out that every feasible relaxed path of graph G^B which represents a feasible relaxed path of the detailed graph has arbitrary long feasible extensions.

$$\begin{aligned}
&\forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{detail,prog,C_i}) : \forall \widehat{p_B} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) : \\
&\quad \widehat{p_B} \vdash_{path} \widehat{p} \;\Rightarrow \\
&\quad \forall n \in \mathbb{N} : \exists \widehat{p_B}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) : \\
&\qquad (\widehat{p_B}, \widehat{p_B}') \in \widehat{\mathit{PrefixOf}}^{B}_{path} \wedge \\
&\qquad \mathit{len}(\widehat{p_B}') = \mathit{len}(\widehat{p_B}) + n
\end{aligned} \tag{A.119}$$

This key statement is a direct consequence of the following hypothesis IH_3 holding for all $i \in \mathbb{N}$.

$$\begin{aligned}
&\mathit{IH}_3(i) \\
\Leftrightarrow &\exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{detail,prog,C_i}) : \\
&\quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}}^{detail,prog,C_i}_{path} \wedge \\
&\quad \exists \widehat{p_B}' \in \widehat{\mathit{RelPaths}}^{B} : \\
&\qquad (\widehat{p_B}, \widehat{p_B}') \in \widehat{\mathit{PrefixOf}}^{B}_{path} \wedge \\
&\qquad \mathit{len}(\widehat{p_B}') \geq \mathit{len}(\widehat{p_B}) + i \wedge \\
&\qquad \widehat{p_B}' \vdash_{path} \widehat{p}'
\end{aligned}$$

The fourth key statement is a slight variant of the third key statement. It states that, for every feasible relaxed path $\widehat{p}$ of the detailed graph which is represented by a feasible relaxed path $\widehat{p_B}$ of G^B and which has arbitrary long feasible extensions that do not guarantee further occurrences of event E, there are arbitrary long extensions of $\widehat{p_B}$ that also do not guarantee more occurrences of event E than $\widehat{p}$ does.

$$\begin{aligned}
&\forall E \in \mathit{Events} : \\
&\quad \forall \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{detail,prog,C_i}) : \forall \widehat{p_B} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) : \\
&\qquad [\widehat{p_B} \vdash_{path} \widehat{p} \wedge \\
&\qquad\quad \forall n \in \mathbb{N} : \exists \widehat{p}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{detail,prog,C_i}) : \\
&\qquad\qquad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}}^{detail,prog,C_i}_{path} \wedge \\
&\qquad\qquad \mathit{len}(\widehat{p}') = \mathit{len}(\widehat{p}) + n \wedge \\
&\qquad\qquad \forall x \in \mathbb{N}_{\geq \mathit{len}(\widehat{p})} \cap \mathbb{N}_{< \mathit{len}(\widehat{p}')} : \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p}', x) = 0] \;\Rightarrow \\
&\qquad \forall n \in \mathbb{N} : \exists \widehat{p_B}' \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) : \\
&\qquad\quad (\widehat{p_B}, \widehat{p_B}') \in \widehat{\mathit{PrefixOf}}^{B}_{path} \wedge \\
&\qquad\quad \mathit{len}(\widehat{p_B}') = \mathit{len}(\widehat{p_B}) + n \wedge \\
&\qquad\quad \sum_{x \in \mathbb{N}_{< \mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p}, x) \geq \sum_{x \in \mathbb{N}_{< \mathit{len}(\widehat{p_B}')}} \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p_B}', x)
\end{aligned} \tag{A.120}$$

This key statement is a direct consequence of the following hypothesis IH_4 holding for all $i \in \mathbb{N}$.

$$\begin{aligned}
&IH_4(i)\\
\Leftrightarrow &\exists \widehat{p}' \in \mathit{PrefExt}(\widehat{p}, E) :\\
&\quad (\widehat{p}, \widehat{p}') \in \widehat{\mathit{PrefixOf}}^{detail,prog,C_i}_{path} \land\\
&\quad \exists \widehat{p_B}' \in \widehat{\mathit{RelPaths}}^B :\\
&\qquad (\widehat{p_B}, \widehat{p_B}') \in \widehat{\mathit{PrefixOf}}^B_{path} \land\\
&\qquad \mathit{len}(\widehat{p_B}') \geq \mathit{len}(\widehat{p_B}) + i \land\\
&\qquad \widehat{p_B}' \vdash_{path} \widehat{p}'
\end{aligned}$$

Now we use the four key statements in order to prove what is actually left to show. It follows from equations (A.110), (A.112), (A.117), (A.119), (6.124), and (6.131) that, for every member t of $\mathit{ExecRuns}^{diverg}_{prog,C_i}$, there is a feasible relaxed path $\widehat{p_B}$ of G^B which upper-bounds the number of event occurrences in t and which ends in a node of $\mathit{Feedback}^B$.

$$\begin{aligned}
&\forall E \in \mathit{Events} :\\
&\quad \forall t \in \mathit{ExecRuns}^{diverg}_{prog,C_i} : \exists \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^B) :\\
&\qquad \mathit{numEvOccur}(prog, C_i, t, E) \leq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{UB}_{prog,C_i,E}(\widehat{p}, x) \land\\
&\qquad \widehat{p}(\mathit{len}(\widehat{p})) \in \mathit{Feedback}^B
\end{aligned} \tag{A.121}$$

Analogously, it follows from equations (A.110), (A.112), (A.118), (A.120), (A.115), (6.124), and (6.131) that, for every member t of $\mathit{ExecRuns}^{diverg,end}_{prog,C_i,E}$, there is a feasible relaxed path $\widehat{p_B}$ of G^B which lower-bounds the number of event occurrences in t and which ends in a node of $\mathit{Feedback}^B$.

$$\begin{aligned}
&\forall E \in \mathit{Events} :\\
&\quad \forall t \in \mathit{ExecRuns}^{diverg,end}_{prog,C_i,E} : \exists \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^B) :\\
&\qquad \mathit{numEvOccur}(prog, C_i, t, E) \geq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{prog,C_i,E}(\widehat{p}, x) \land\\
&\qquad \widehat{p}(\mathit{len}(\widehat{p})) \in \mathit{Feedback}^B
\end{aligned} \tag{A.122}$$

Moreover, due to the soundness of property lifting, the set of feasible relaxed paths of G^B which end in a node of $\mathit{Feedback}^B$ is a subset of $\widehat{\mathit{LessFeedPaths}}^B_{prog,C_i}$.

$$\{\widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^B) \mid \widehat{p}(\mathit{len}(\widehat{p})) \in \mathit{Feedback}^B\} \underset{\substack{(6.125)\\(6.126)}}{\subseteq} \widehat{\mathit{LessFeedPaths}}^B_{prog,C_i} \tag{A.123}$$

Finally, statement (A.106) is a consequence of statements (A.121) and (A.123). In the same way, statement (A.107) is a consequence of statements (A.122) and (A.123). □

Proof of Statement 6.137. In this proof, we show that the following statement holds, which directly implies statement (6.137).

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad G^B \models G^{\mathit{detail},\mathit{prog},C_i} \Rightarrow\\ &\quad \forall t \in \mathit{ExecRuns}^{\mathit{min\text{-}relev}}_{\mathit{prog},C_i,E} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^{B}_{\mathit{prog},C_i,E} :\\ &\qquad \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) \geq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{\mathit{prog},C_i,E}(\widehat{p}, x)\end{aligned} \tag{A.124}$$

Most of this proof is identical to the previous proof of statement (6.130). Thus, we only present the differences. In addition to what has already been done in the previous proof, we only have to prove the following statement.

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad G^B \models G^{\mathit{detail},\mathit{prog},C_i} \Rightarrow\\ &\quad \forall t \in \mathit{ExecRuns}^{\mathit{diverg},\mathit{end}}_{\mathit{prog},C_i,E} : \exists \widehat{p} \in \widehat{\mathit{LessFeedPaths}}^{B}_{\mathit{prog},C_i,E} :\\ &\qquad \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) \geq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{\mathit{prog},C_i,E}(\widehat{p}, x)\end{aligned} \tag{A.125}$$

It follows from equations (A.110), (A.112), (A.118), (A.120), (A.115), (6.133), (6.131), and the subsumption of the detailed graph by G^B that, for every member t of $\mathit{ExecRuns}^{\mathit{diverg},\mathit{end}}_{\mathit{prog},C_i,E}$, there is a feasible relaxed path $\widehat{p_B}$ of G^B which lower-bounds the number of event occurrences in t and which ends in a node of $\mathit{Feedback}^B_E$.

$$\begin{aligned}&\forall E \in \mathit{Events} :\\ &\quad \forall t \in \mathit{ExecRuns}^{\mathit{diverg},\mathit{end}}_{\mathit{prog},C_i,E} : \exists \widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) :\\ &\quad \mathit{numEvOccur}(\mathit{prog}, C_i, t, E) \geq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{p})}} \widehat{\mathit{wEvent}}^{LB}_{\mathit{prog},C_i,E}(\widehat{p}, x) \land\\ &\quad \widehat{p}(\mathit{len}(\widehat{p})) \in \mathit{Feedback}^B_E\end{aligned} \tag{A.126}$$

Moreover, due to the soundness of property lifting, the set of feasible relaxed paths of G^B which end in a node of $\mathit{Feedback}^B_E$ is a subset of $\widehat{\mathit{LessFeedPaths}}^{B}_{\mathit{prog},C_i,E}$.

$$\{\widehat{p} \in \mathit{feas}(\widehat{\mathit{RelPaths}}^{B}) \mid \widehat{p}(\mathit{len}(\widehat{p})) \in \mathit{Feedback}^B_E\} \underset{\substack{(6.134)\\(6.135)}}{\subseteq} \widehat{\mathit{LessFeedPaths}}^{B}_{\mathit{prog},C_i,E} \tag{A.127}$$

Finally, statements (A.126) and (A.127) imply statement (A.125). □

Proof Sketch for Statement 6.154. In order to prove that the constructed graph subsumes the detailed graph (i.e. $G^{\mathit{constr}} \models G^{\mathit{detail},\mathit{prog},C_i}$), we have to show that criteria (6.C7) to (6.C10) hold.

For this proof, we define the representation relation $\vdash_{path}$ in the following way. Remember that, so far, we only required the existence of this relation (about which criteria (6.C8) to (6.C10) argue).

$$\begin{aligned}
&\forall \widehat{p_A} \in \widehat{SubPaths}^{detail,prog,C_i} : \forall \widehat{p_B} \in \widehat{SubPaths}^{constr} : \\
&\quad \widehat{p_B} \vdash_{path} \widehat{p_A} \\
&\Leftrightarrow \exists part \in Partitionings(len(\widehat{p_B}), len(\widehat{p_A})) : \\
&\qquad [\forall x \in \mathbb{N}_{<len(\widehat{p_B})} : \\
&\qquad\quad \widehat{p_A}(from(part,x)) \in \widehat{p_B}(x)] \wedge \\
&\qquad \widehat{p_A}(len(\widehat{p_A})) \in \widehat{p_B}(len(\widehat{p_B})) \wedge \\
&\qquad \forall E \in Events : \\
&\qquad\quad \forall x \in \mathbb{N}_{<len(\widehat{p_B})} : \\
&\qquad\qquad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{wE^{UB}}(\widehat{p_A}, i) \leq \widehat{wE^{UB}}(\widehat{p_B}, x) \wedge \\
&\qquad\qquad \sum_{from(part,x) \leq i \leq to(part,x)} \widehat{wE^{LB}}(\widehat{p_A}, i) \geq \widehat{wE^{LB}}(\widehat{p_B}, x)
\end{aligned} \tag{A.128}$$

According to equation (6.120), relation $\vdash_{path}$ implies path subsumption ($\vDash_{path}$). Thus, criterion (6.C8) holds.

It follows from $\vdash_{path}$ and Algorithms 6.1 to 6.3 that every path from a start node to an end node of the detailed graph is represented by a path from a start node to an end node of the constructed graph. Since path representation implies path subsumption (criterion (6.C8)), every path from a start node to an end node of the detailed graph is subsumed by a path from a start node to an end node of the constructed graph. Thus, criterion (6.C7) holds.

Next, it follows from $\vdash_{path}$ and the definition of the start nodes of the constructed graph in Algorithm 6.1 that criterion (6.C9) holds.

Finally, we show criterion (6.C10) by using $\vdash_{path}$, equation (6.140), and Algorithms 6.1 to 6.3. □

Proof that Property P_{wc} Can Safely Be Lifted to $\widehat{P_{wc}}$. System property P_{wc} implies a slightly relaxed property $P_{wc'}$. In the following, let C_i be the particular processor core for which want to calculate a WCET bound.

$$\begin{aligned}
&\quad P_{wc}(t) \\
&\underset{(7.14)}{\Leftrightarrow} [\forall C_k \in Cores : \forall x \in \mathbb{N}_{<len(t)} : \\
&\qquad Blocked_{C_k}(t,x) \Rightarrow \exists C_j \in Cores \setminus \{C_k\} : Granted_{C_j}(t,x)] \\
&\Rightarrow\ [\forall x \in \mathbb{N}_{\leq len(t)} : \sum_{y \in \mathbb{N}_{<x}} Blocked_{C_i}(t,y) \leq \sum_{C_j \in Cores \setminus \{C_i\}} \sum_{y \in \mathbb{N}_{<x}} Granted_{C_j}(t,y)] \\
&\Leftrightarrow:\ P_{wc'}(t)
\end{aligned} \tag{A.129}$$

The relaxed property $P_{wc'}$ is already relatively close to the lifted version $\widehat{P_{wc}}$. However, the nested $\sum$ symbols in property $P_{wc'}$ are not directly supported by our lifting rules (cf. Table 5.1). Thus, instead, we directly prove that soundness criterion (4.C1) is fulfilled.

$$
\begin{aligned}
& [\exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) : P_{wc}(t)] \\
\underset{(A.129)}{\Rightarrow}\; & [\exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) : P_{wc'}(t)] \\
\underset{(A.129)}{\Leftrightarrow}\; & [\exists t \in \gamma_{trace,prog,C_i}(\widehat{t}) : \\
& \qquad \forall x \in \mathbb{N}_{\leq len(t)} : \sum_{y \in \mathbb{N}_{<x}} Blocked_{C_i}(t, y) \leq \sum_{C_j \in Cores \setminus \{C_i\}} \; \sum_{y \in \mathbb{N}_{<x}} Granted_{C_j}(t, y)] \\
\underset{\substack{(5.32)\\(5.41)\\(7.21)\\(7.24)}}{\Rightarrow}\; & \forall x \in \mathbb{N}_{\leq len(\widehat{t})} : \sum_{y \in \mathbb{N}_{<x}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t}, y) \leq \sum_{C_j \in Cores \setminus \{C_i\}} \; \sum_{y \in \mathbb{N}_{<x}} \widehat{Granted}^{UB}_{C_j}(\widehat{t}, y) \\
\underset{(7.25)}{\Leftrightarrow}\; & \widehat{P_{wc}}(\widehat{t}) \quad \square
\end{aligned}
$$

Proof that the Formal Setup of Equations 7.27, 7.28, and 7.29 Fulfills All Criteria Required by Section 4.2.3. The formal setup of equations (7.27), (7.28), and (7.29) corresponds to the setup presented in Section 4.2.3. However, the updates of the approximation variables (equations (7.28), and (7.29)) are presented in a less modular fashion than in Section 4.2.3. Thus, first, we modularize the notation of the updates of the approximation variables accordingly.

$$\forall C_k \in Cores : \widehat{Approx}^{C_k} \leftarrow F^{C_k}(\overrightarrow{Approx_{\widehat{C_k}}}) \tag{A.130}$$

$$\forall C_k \in Cores : F^{C_k}(\overrightarrow{Approx_{\widehat{C_k}}}) = \{\widehat{t^{C_k}} \in \widehat{ExecRuns}^{C_k''}_{prog,C_i} \mid \widetilde{P^{C_k}}(\widehat{t^{C_k}}, \overrightarrow{Approx_{\widehat{C_k}}})\} \tag{A.131}$$

$$
\begin{aligned}
& \widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{Approx_{\widehat{C_i}}}) \\
\Leftrightarrow & \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \\
& \qquad \leq \sum_{C_j \in Cores \setminus \{C_i\}} \; \max_{\widehat{t^{C_j}} \in \widehat{Approx}^{C_j}} \; \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x)
\end{aligned}
\tag{A.132}
$$

$$
\begin{aligned}
& \forall C_j \in Cores \setminus \{C_i\} : \\
& \widetilde{P^{C_j}}(\widehat{t^{C_j}}, \overrightarrow{Approx_{\widehat{C_j}}}) \\
\Leftrightarrow & \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle}^{LB}(\widehat{t^{C_j}}, x) \leq \max_{\widehat{t^{C_i}} \in \widehat{Approx}^{C_i}} \; \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle}^{UB}(\widehat{t^{C_i}}, x)
\end{aligned}
\tag{A.133}
$$

According to equations (7.22) and (7.26), the following property $\widehat{P}$ holds for all members of set $\widehat{Cmpnd''}$, which we plan to overapproximate iteratively.

$$\widehat{P}(\widehat{t}) \Leftrightarrow \widehat{P_{wc}}(\widehat{t}) \land \exists n \in \mathbb{N} : \forall C_k \in Cores : len(\pi^{C_k}_{trace}(\widehat{t})) = n \tag{A.134}$$

$$\forall \widehat{t} \in \widehat{Cmpnd''} : \widehat{P}(\widehat{t}) \tag{A.135}$$

With this information in place, we can show that the $\widetilde{P^{C_k}}$ fulfill *soundness criterion* (4.C2) with respect to $\widehat{P}$. This means we have to show the following statement.

$$
\begin{aligned}
&\forall C_k \in \mathit{Cores} : \forall \widehat{t^{C_k}} \in \widehat{\mathit{ExecRuns}}^{C_k\prime\prime}_{prog,C_i} : \forall \overrightarrow{\mathit{Approx}_{\overline{C_k}}} : \\
&\qquad [\exists (\widehat{t^{C_1}}, \dots, \widehat{t^{C_{k-1}}}, \widehat{t^{C_{k+1}}}, \dots, \widehat{t^{C_{|Cores|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_k}}}) : \\
&\qquad\quad \widehat{P}(\widehat{t^{C_1}}, \dots, \widehat{t^{C_{k-1}}}, \widehat{t^{C_k}}, \widehat{t^{C_{k+1}}}, \dots, \widehat{t^{C_{|Cores|}}})] \\
&\quad \Rightarrow \widetilde{P^{C_k}}(\widehat{t^{C_k}}, \overrightarrow{\mathit{Approx}_{\overline{C_k}}})
\end{aligned}
\tag{A.136}
$$

We prove statement (A.136) in two steps. First, we prove the part of it which argues about core C_i.

$$
\begin{aligned}
&\forall \widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} : \forall \overrightarrow{\mathit{Approx}_{\overline{C_i}}} : \\
&\qquad [\exists (\widehat{t^{C_1}}, \dots, \widehat{t^{C_{i-1}}}, \widehat{t^{C_{i+1}}}, \dots, \widehat{t^{C_{|Cores|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_i}}}) : \\
&\qquad \widehat{P}(\widehat{t^{C_1}}, \dots, \widehat{t^{C_{i-1}}}, \widehat{t^{C_i}}, \widehat{t^{C_{i+1}}}, \dots, \widehat{t^{C_{|Cores|}}})] \\
\underset{\substack{(A.134)\\(4.18)}}{\Leftrightarrow}\ & [\exists (\widehat{t^{C_1}}, \dots, \widehat{t^{C_{i-1}}}, \widehat{t^{C_{i+1}}}, \dots, \widehat{t^{C_{|Cores|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_i}}}) : \\
&\ \widehat{P_{wc}}(\widehat{t^{C_1}}, \dots, \widehat{t^{C_{i-1}}}, \widehat{t^{C_i}}, \widehat{t^{C_{i+1}}}, \dots, \widehat{t^{C_{|Cores|}}}) \wedge \\
&\ \exists n \in \mathbb{N} : \forall C_k \in \mathit{Cores} : \mathit{len}(\widehat{t^{C_k}}) = n] \\
\underset{\substack{(7.24)\\(7.25)}}{\Rightarrow}\ & [\exists (\widehat{t^{C_1}}, \dots, \widehat{t^{C_{i-1}}}, \widehat{t^{C_{i+1}}}, \dots, \widehat{t^{C_{|Cores|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_i}}}) : \\
&\ \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in Cores \setminus \{C_i\}} \ \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x) \wedge \\
&\ \exists n \in \mathbb{N} : \forall C_k \in \mathit{Cores} : \mathit{len}(\widehat{t^{C_k}}) = n] \\
\Rightarrow\ & [\exists (\widehat{t^{C_1}}, \dots, \widehat{t^{C_{i-1}}}, \widehat{t^{C_{i+1}}}, \dots, \widehat{t^{C_{|Cores|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_i}}}) : \\
&\ \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq \sum_{C_j \in Cores \setminus \{C_i\}} \ \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x)] \\
\Leftrightarrow\ & \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \\
&\quad \leq \sum_{C_j \in Cores \setminus \{C_i\}} \ \max_{\widehat{t^{C_j}} \in \widehat{Approx^{C_j}}} \ \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x) \\
\underset{(A.132)}{\Leftrightarrow}\ & \widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{\mathit{Approx}_{\overline{C_i}}})
\end{aligned}
$$

Next, we prove the part of statement (A.136) which argues about the concurrent cores of core C_i.

$$
\begin{aligned}
&\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \forall \widehat{t^{C_j}} \in \widehat{\mathit{ExecRuns}}^{C_j\prime\prime}_{prog,C_i} : \forall \overrightarrow{\mathit{Approx}_{\overline{C_j}}} : \\
&\qquad [\exists (\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_j}}}) : \\
&\qquad\quad \widehat{P}(\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_j}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}})] \\
\underset{\substack{(A.134)\\(4.18)}}{\Leftrightarrow}\ &\qquad [\exists (\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_j}}}) : \\
&\qquad\quad \widehat{P_{wc}}(\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_j}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}}) \wedge \\
&\qquad\quad \exists n \in \mathbb{N} : \forall C_k \in \mathit{Cores} : \mathit{len}(\widehat{t^{C_k}}) = n] \\
\Rightarrow\ &\qquad [\exists (\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_j}}}) : \\
&\qquad\quad \exists n \in \mathbb{N} : \forall C_k \in \mathit{Cores} : \mathit{len}(\widehat{t^{C_k}}) = n] \\
\Rightarrow\ &\qquad [\exists (\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_j}}}) : \\
&\qquad\quad \mathit{len}(\widehat{t^{C_j}}) = \mathit{len}(\widehat{t^{C_i}})] \\
\Rightarrow\ &\qquad [\exists (\widehat{t^{C_1}}, \ldots, \widehat{t^{C_{j-1}}}, \widehat{t^{C_{j+1}}}, \ldots, \widehat{t^{C_{|\mathit{Cores}|}}}) \in \chi(\overrightarrow{\mathit{Approx}_{\overline{C_j}}}) : \\
&\qquad\quad \mathit{len}(\widehat{t^{C_j}}) \leq \mathit{len}(\widehat{t^{C_i}})] \\
\Leftrightarrow\ &\qquad \mathit{len}(\widehat{t^{C_j}}) \leq \max_{\widehat{t^{C_i}} \in \widehat{\mathit{Approx}^{C_i}}} \mathit{len}(\widehat{t^{C_i}}) \\
\underset{(5.40)}{\Leftrightarrow}\ &\qquad \sum_{x \in \mathbb{N}_{< \mathit{len}(\widehat{t^{C_j}})}} \widehat{\mathit{Cycle}^{LB}}(\widehat{t^{C_j}}, x) \leq \max_{\widehat{t^{C_i}} \in \widehat{\mathit{Approx}^{C_i}}} \sum_{x \in \mathbb{N}_{< \mathit{len}(\widehat{t^{C_i}})}} \widehat{\mathit{Cycle}^{UB}}(\widehat{t^{C_i}}, x) \\
\underset{(A.133)}{\Leftrightarrow}\ &\qquad \widetilde{P^{C_j}}(\widehat{t^{C_j}}, \overrightarrow{\mathit{Approx}_{\overline{C_j}}})
\end{aligned}
$$

This concludes the proof of statement (A.136). As a consequence of statement (A.136), soundness criterion (4.C2) is fulfilled.

Moreover, we can show that the $\widetilde{P^{C_k}}$ fulfill *monotonicity criterion* (4.C3). This means we have to show the following statement.

$$
\begin{aligned}
&\forall C_k \in \mathit{Cores} : \forall \widehat{t^{C_k}} \in \widehat{\mathit{ExecRuns}}^{C_k\prime\prime}_{prog,C_i} : \forall \overrightarrow{\mathit{Approx}_{\overline{C_k}}}, \overrightarrow{\mathit{Approx}'_{\overline{C_k}}} : \\
&\quad \overrightarrow{\mathit{Approx}'_{\overline{C_k}}} \subseteq_{pairwise} \overrightarrow{\mathit{Approx}_{\overline{C_k}}} \\
&\quad \Rightarrow [\widetilde{P^{C_k}}(\widehat{t^{C_k}}, \overrightarrow{\mathit{Approx}'_{\overline{C_k}}}) \Rightarrow \widetilde{P^{C_k}}(\widehat{t^{C_k}}, \overrightarrow{\mathit{Approx}_{\overline{C_k}}})]
\end{aligned}
\tag{A.137}
$$

We prove statement (A.137) in two steps, too. First, we prove the part of it which argues about core C_i.

$$\begin{aligned}
&\forall \widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} : \forall \overrightarrow{Approx_{\overline{C_i}}}, \overrightarrow{Approx'_{\overline{C_i}}} : \\
&\qquad \overrightarrow{Approx'_{\overline{C_i}}} \subseteq_{pairwise} \overrightarrow{Approx_{\overline{C_i}}} \\
&\Leftrightarrow \quad [\forall C_j \in Cores \setminus \{C_i\} : \widehat{Approx'}^{C_j} \subseteq \widehat{Approx}^{C_j}] \\
&\Rightarrow \quad [\forall C_j \in Cores \setminus \{C_i\} : \\
&\qquad \max_{\widehat{t^{C_j}} \in \widehat{Approx'}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x) \\
&\qquad\quad \leq \max_{\widehat{t^{C_j}} \in \widehat{Approx}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x)] \\
&\underset{(A.132)}{\Rightarrow} [\widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{Approx'_{\overline{C_i}}}) \Rightarrow \widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{Approx_{\overline{C_i}}})]
\end{aligned}$$

Next, we prove the part of statement (A.137) which argues about the concurrent cores of core C_i.

$$\begin{aligned}
&\forall C_j \in Cores \setminus \{C_i\} : \forall \widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i} : \forall \overrightarrow{Approx_{\overline{C_j}}}, \overrightarrow{Approx'_{\overline{C_j}}} : \\
&\qquad \overrightarrow{Approx'_{\overline{C_j}}} \subseteq_{pairwise} \overrightarrow{Approx_{\overline{C_j}}} \\
&\Leftrightarrow \quad [\forall C_k \in Cores \setminus \{C_j\} : \widehat{Approx'}^{C_k} \subseteq \widehat{Approx}^{C_k}] \\
&\underset{C_i \neq C_j}{\Rightarrow} \quad \widehat{Approx'}^{C_i} \subseteq \widehat{Approx}^{C_i}] \\
&\Rightarrow \quad [\max_{\widehat{t^{C_i}} \in \widehat{Approx'}^{C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle}^{UB}(\widehat{t^{C_i}}, x) \\
&\qquad\quad \leq \max_{\widehat{t^{C_i}} \in \widehat{Approx}^{C_i}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle}^{UB}(\widehat{t^{C_i}}, x)] \\
&\underset{\substack{C_i \neq C_j \\ (A.133)}}{\Rightarrow} \quad [\widetilde{P^{C_j}}(\widehat{t^{C_j}}, \overrightarrow{Approx'_{\overline{C_j}}}) \Rightarrow \widetilde{P^{C_j}}(\widehat{t^{C_j}}, \overrightarrow{Approx_{\overline{C_j}}})]
\end{aligned}$$

This concludes the proof of statement (A.137). As a consequence of statement (A.137), monotonicity criterion (4.C3) is fulfilled. Thus, all criteria required by Section 4.2.3 are fulfilled. □

Counter Example Demonstrating that, in General, the First Iterative Overapproximation Approach Presented in Section 7.5 Is Not Sound in Combination with an Obvious Optimistic Initialization. The first iterative approach presented in Section 7.5 starts from a maximally pessimistic initialization—following Section 4.2.3. During the derivation of the approach, we also experimented with the following optimistic initialization—following Section 4.2.4.

$$\widehat{Approx}^{C_i} \leftarrow \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i''}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\} \tag{A.138}$$

$$\forall C_j \in Cores \setminus \{C_i\} : \widehat{Approx}^{C_j} \leftarrow \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j''}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle}^{LB}(\widehat{t^{C_j}}, x) \leq 0\} \tag{A.139}$$

It initializes the approximation variable of core C_i to only those members of $\widehat{ExecRuns}^{C_i''}_{prog,C_i}$ that do not guarantee any blocked cycles at the shared bus. The approximation variable of a concurrent core C_j, in contrast, is initialized to the members of $\widehat{ExecRuns}^{C_j''}_{prog,C_i}$ of length zero. Note that this style of initialization is very similar to the updates of the approximation variables (cf. equations (7.28), and (7.29))—with the exception that the maximum-calculations based on other approximation variables are replaced by optimistically assuming their respective values to be zero.

Following this optimistic initialization, Algorithm 7.1 starts by assuming that the program on core C_i is not blocked at all at the shared bus. In an iterative manner, it allows more and more blocked cycles in dependence on the arrival curve values of the concurrent processor cores.

Unfortunately, in general, the resulting formal setup (i.e. equations (A.138), (A.139), (7.28), and (7.29)) does not fulfill all criteria required by Section 4.2.4. Figure A.1 provides a *minimal counter-example* demonstrating that a fixed point reached from this optimistic initialization can be unsound with respect to the concrete traces. For the counter-example, we consider a very simple dual-core processor without instruction pipelining. The processor shall feature a shared bus. Access requests by the processor cores (C_1 and C_2) shall be arbitrated according to the Round-Robin policy. The particular flavor of the Round-Robin policy implemented on the hardware platform shall prefer core C_1 during arbitration in case both cores have not yet been granted access to the bus since system start. The considered example program requests a single time unit of granted access to the bus. The program terminates while performing this granted access. Both processor cores shall execute this program immediately after system start. Afterwards, they shall not execute any other programs requesting access to the shared bus. As a consequence of the bus arbitration, on the concrete system, core C_1 is granted (■) access to the bus first. Thus, core C_2 is blocked (⊠) for one time unit before being granted access to the bus. This means that the example program has a worst-case execution time of two time units when executed on core C_2. Note that Figure A.1 only presents the full program execution runs—for the concrete system as well as for the level of approximation based on sequences of abstract states. For the sake of readability, the respective prefixes are omitted.

At the level of approximation, we choose a setup for the calculation of a WCET bound for the example program executed on core C_2. First, we consider a case in which the optimistic initialization leads to a fixed point which safely overapproximates the concrete traces. The abstract trace set of each core also contains one infeasible abstract trace and all its prefixes. These infeasible abstract traces assume that core C_2 is granted access first, which cannot happen on the concrete system. According to the optimistic initialization, the approximation variable of core C_1 initially only contains prefixes of length zero (denoted by ε, cf. equation (A.139)). The approximation variable of core C_2 initially contains prefixes of length zero and an abstract trace of length one (cf. equation (A.138)). In the following, we present a fixed point iteration (following equations (7.28), and (7.29)) based on this initialization. For the sake of simplicity, it assumes a simultaneous update of both approximation variables.

Operation of the Concrete System:

C_1: ■
C_2: ⊠■

Abstract Models *with* Infeasible Abstract Traces:

C_1: ■, ⊠■
C_2: ■, ⊠■

Abstract Models *without* Infeasible Abstract Traces:

C_1: ■
C_2: ⊠■

Figure A.1.: Counter-example demonstrating the potential unsoundness of a fixed point reached from the optimistic initialization (cf. equations (A.138) and (A.139)).

iteration	$\widehat{Approx}^{C_1}$	$\widehat{Approx}^{C_2}$
init	{ε}	{ε, ■}
1	{ε, ■, ⊠}	{ε, ■}
2	{ε, ■, ⊠}	{ε, ■, ⊠, ⊠■}
3	{ε, ■, ⊠, ⊠■}	{ε, ■, ⊠, ⊠■}
4	{ε, ■, ⊠, ⊠■}	{ε, ■, ⊠, ⊠■}

The resulting fixed point safely overapproximates the concrete traces and, thus, results in a safe WCET bound of two time units for the example program executed on core C_2.

Next, we only take into account the feasible abstract traces and their prefixes. This means that the approximation variable of core C_2 initially also only contains prefixes of length zero (cf. equation (A.138)). As a consequence, the initialization is already a fixed point.

iteration	$\widehat{Approx}^{C_1}$	$\widehat{Approx}^{C_2}$
init	{ε}	{ε}
1	{ε}	{ε}

The obtained fixed point, however, results in a WCET bound of zero time units for the example program executed on core C_2, which is unsound with respect to the concrete traces. This concludes the counter-example demonstrating the potential unsoundness of a fixed point reached from the proposed optimistic initialization.

Thus, in general, the formal setup of equations (A.138), (A.139), (7.28), and (7.29) does not fulfill all criteria required by Section 4.2.4. The intuitive problem is that a particular sequence of abstract states of length x which is blocked for the y^{th} time at its last position can only be added to $\widehat{Approx}^{C_i}$ if there is already a member of $\widehat{Approx}^{C_i}$ which has a length of at least x and is blocked less than y times. In order to avoid this problem, we can rely on the following additional assumption about the set $\widehat{ExecRuns}^{C_i''}_{prog,C_i}$, from which the members of $\widehat{Approx}^{C_i}$ are drawn (cf. equations (A.138) and (7.29)).

$$\begin{aligned}
&\forall \widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i''}_{prog,C_i} :\\
&\quad [len(\widehat{t^{C_i}}) \geq 1 \wedge \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, len(\widehat{t^{C_i}}) - 1)]\\
&\quad \Rightarrow \exists \widehat{t^{C_i\prime}} \in \widehat{ExecRuns}^{C_i''}_{prog,C_i} :\\
&\qquad len(\widehat{t^{C_i\prime}}) \geq len(\widehat{t^{C_i}}) \wedge\\
&\qquad \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i\prime}})}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i\prime}}, x) < \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x)
\end{aligned} \tag{A.140}$$

Under the additional assumption (A.140), we can prove that the formal setup of equations (A.138), (A.139), (7.28), and (7.29) is guaranteed to fulfill all criteria required by Section 4.2.4. A formal proof, however, is omitted due to space and time constraints.

While our pessimistic baseline model of the shared-bus interference (i.e. $\widehat{ExecRuns}^{C_i}_{prog,C_i}$, cf. Section 7.2) intuitively fulfills assumption (A.140), the pruning of provably infeasible members of the baseline model may invalidate this assumption (cf. previous counter-example). Thus, in order to safely guarantee assumption (A.140), we would have to prove that the overall set of lifted system properties used in the definition of $\widehat{ExecRuns}^{C_i''}_{prog,C_i}$ (cf. equations (7.16) and (7.17)) does not invalidate assumption (A.140). It is, however, unclear how to conduct such a proof. Moreover, it is not in the spirit of property lifting to additionally check whether the overall set of lifted properties fulfills a non-trivial consistency criterion with respect to the set of abstract traces. □

Proof that under Assumption 7.33 the Formal Setup of Equations 7.30, 7.31, 7.28, and 7.32 Fulfills All Criteria Required by Section 4.2.4. The formal setup of equations (7.30), (7.31), (7.28), and (7.32) corresponds to the setup presented in Section 4.2.4. The updates of the approximation variables of the concurrent cores C_j of C_i (equation (7.28)) shall be specified in the modularized notation presented in the previous proof (equations (A.130), (A.131), and (A.133)). The correspondingly modularized notation for the update of the approximation variable of core C_i id completed by the following predicate.

$$\begin{aligned}
&\widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{Approx_{\overline{C_i}}})\\
\Leftrightarrow &\sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x)\\
&\quad \leq \sum_{C_j \in Cores \setminus \{C_i\}} \max_{\widehat{t^{C_j}} \in \widehat{Approx}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x)
\end{aligned} \tag{A.141}$$

In the previous proof, we have already shown that the $\widetilde{P^{C_j}}$ fulfill *monotonicity criterion* (4.C3) for all $C_j \neq C_i$. Now, we prove the same for the $\widetilde{P^{C_i}}$ just specified.

$$
\begin{aligned}
&\forall \widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} : \forall \overrightarrow{\mathit{Approx}_{\overline{C_i}}}, \overrightarrow{\mathit{Approx}'_{\overline{C_i}}} : \\
&\qquad \overrightarrow{\mathit{Approx}'_{\overline{C_i}}} \subseteq_{pairwise} \overrightarrow{\mathit{Approx}_{\overline{C_i}}} \\
&\quad \Leftrightarrow \quad [\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \widehat{\mathit{Approx}'}^{C_j} \subseteq \widehat{\mathit{Approx}}^{C_j}] \\
&\quad \Rightarrow \quad [\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \\
&\qquad\qquad \max_{\widehat{t^{C_j}} \in \widehat{\mathit{Approx}'}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x) \\
&\qquad\qquad\quad \leq \max_{\widehat{t^{C_j}} \in \widehat{\mathit{Approx}}^{C_j}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x)] \\
&\quad \underset{(A.141)}{\Rightarrow} [\widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{\mathit{Approx}'_{\overline{C_i}}}) \Rightarrow \widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{\mathit{Approx}_{\overline{C_i}}})]
\end{aligned}
$$

The optimistic initialization (equations (7.30) and (7.31)) corresponds to the following initialization values.

$$\widehat{\mathit{Init}}^{C_i} = \{\widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\} \tag{A.142}$$

$$\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \widehat{\mathit{Init}}^{C_j} = \{\widehat{t^{C_j}} \in \widehat{\mathit{ExecRuns}}^{C_j\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{\mathit{Cycle}}^{LB}(\widehat{t^{C_j}}, x) \leq 0\} \tag{A.143}$$

Note that we assume program *prog* to be actually executed on core C_i (i.e. $\mathit{ExecRuns}_{prog,C_i} \neq \emptyset$). As a consequence, $\widehat{\mathit{ExecRuns}}^{C_i\prime\prime}_{prog,C_i}$ and the $\widehat{\mathit{ExecRuns}}^{C_j\prime\prime}_{prog,C_i}$ for $C_j \neq C_i$ must also not be empty. It follows that each of the above initialization values must at least contain a prefix of length zero and, thus, cannot be empty.

$$\forall C_k \in \mathit{Cores} : \widehat{\mathit{Init}}^{C_k} \neq \emptyset \tag{A.144}$$

With this information in place, we can show that the update functions F^{C_k} fulfill criterion (4.C4). This means we have to show the following statement.

$$\forall C_k \in \mathit{Cores} : F^{C_k}(\overrightarrow{\mathit{Init}_{\overline{C_k}}}) \supseteq \widehat{\mathit{Init}}^{C_k} \tag{A.145}$$

The part of statement (A.145) which argues about core C_i is proven as follows.

$$
\begin{aligned}
& F^{C_i}(\overrightarrow{Init_{\overline{C_i}}}) \\
\underset{(A.131)}{=}\ & \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} \mid \widetilde{P^{C_i}}(\widehat{t^{C_i}}, \overrightarrow{Init_{\overline{C_i}}})\} \\
\underset{(A.141)}{=}\ & \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \\
& \qquad \le \sum_{C_j \in Cores \setminus \{C_i\}} \max_{\widehat{t^{C_j}} \in \widehat{Init^{C_j}}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j}}, x)\} \\
\underset{\substack{(A.143)\\(A.144)}}{=}\ & \{\widehat{t^{C_i}} \in \widehat{ExecRuns}^{C_i\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \le 0\} \\
\underset{(A.142)}{=}\ & \widehat{Init^{C_i}}
\end{aligned}
$$

The part of statement (A.145) which argues about the concurrent cores of core C_i is proven as follows.

$$
\begin{aligned}
& \forall C_j \in Cores \setminus \{C_i\} : \\
& \qquad F^{C_j}(\overrightarrow{Init_{\overline{C_j}}}) \\
& \underset{(A.131)}{=}\ \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i} \mid \widetilde{P^{C_j}}(\widehat{t^{C_j}}, \overrightarrow{Init_{\overline{C_j}}})\} \\
& \underset{(A.133)}{=}\ \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle^{LB}}(\widehat{t^{C_j}}, x) \\
& \qquad \le \max_{\widehat{t^{C_i}} \in \widehat{Init^{C_i}}} \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})}} \widehat{Cycle^{UB}}(\widehat{t^{C_i}}, x)\} \\
& \underset{\substack{(A.142)\\(A.144)}}{\supseteq}\ \{\widehat{t^{C_j}} \in \widehat{ExecRuns}^{C_j\prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_j}})}} \widehat{Cycle^{LB}}(\widehat{t^{C_j}}, x) \le 0\} \\
& \underset{(A.143)}{=}\ \widehat{Init^{C_j}}
\end{aligned}
$$

Next, we prove that—under assumption (7.33)—the formal setup of equations (7.30), (7.31), (7.28), and (7.32) fulfills *soundness criterion* (4.C5). This means that we prove the following statement.

$$
\begin{aligned}
& \forall C_k \in Cores : \forall \widehat{t^{C_k}} \in \pi^{C_k}(\widehat{Cmpnd''}) : \\
& \quad \exists updSequ \in UpdateSequ : \widehat{t^{C_k}} \in \pi^{C_k}(\chi(apply(\overrightarrow{Init}, updSequ)))
\end{aligned} \tag{A.146}
$$

To this end, first, we prove the part of statement (A.146) which argues about core C_i.

$$
\begin{aligned}
& \forall \widehat{t^{C_i}} \in \pi^{C_i}(\widehat{Cmpnd''}) : \\
& \quad \exists updSequ \in UpdateSequ : \widehat{t^{C_i}} \in \pi^{C_i}(\chi(apply(\overrightarrow{Init}, updSequ)))
\end{aligned} \tag{A.147}
$$

As a preparation for the inductive proof of statement (A.147), we define one subset of $\widehat{\mathit{Cmpnd}}''$ per natural number n as follows.

$$\forall n \in \mathbb{N} : \widehat{\mathit{Cmpnd}}''_n = \{(\dots, \widehat{t^{C_i}}, \dots) \in \widehat{\mathit{Cmpnd}}'' \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq n\} \tag{A.148}$$

Note that the union over the sets $\widehat{\mathit{Cmpnd}}''_n$ for all natural number n results again in the set $\widehat{\mathit{Cmpnd}}''$.

$$\widehat{\mathit{Cmpnd}}'' = \bigcap_{n \in \mathbb{N}} \widehat{\mathit{Cmpnd}}''_n \tag{A.149}$$

Thus, an inductive proof that the following hypothesis $\mathit{IH}(n)$ holds for all natural numbers n is at the same time a proof of statement (A.147).

$$\begin{aligned}
&\mathit{IH}(n) \\
\Leftrightarrow &\forall \widehat{t^{C_i}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}}''_n) : \\
&\quad \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_i}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))
\end{aligned} \tag{A.150}$$

For the *induction start* (i.e. $\mathit{IH}(0)$), we begin by proving that the set $\pi^{C_i}(\widehat{\mathit{Cmpnd}}''_0)$ is contained in $\widehat{\mathit{Init}^{C_i}}$.

$$\begin{aligned}
&\pi^{C_i}(\widehat{\mathit{Cmpnd}}''_0) \\
\underset{(A.148)}{=}\ &\pi^{C_i}(\{(\dots, \widehat{t^{C_i}}, \dots) \in \widehat{\mathit{Cmpnd}}'' \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\}) \\
\underset{\substack{(7.26)\\(7.22)\\(7.20)}}{=}\ &\pi^{C_i}(\{(\dots, \widehat{t^{C_i}}, \dots) \in \widehat{\mathit{ExecRuns}}^{C_1 \prime\prime}_{prog,C_i} \times \widehat{\mathit{ExecRuns}}^{C_2 \prime\prime}_{prog,C_i} \times \dots \mid \\
&\qquad [\exists m \in \mathbb{N} : \forall C_k \in \mathit{Cores} : len(\widehat{t^{C_k}}) = m] \wedge \widehat{P_{wc}}((\dots, \widehat{t^{C_i}}, \dots)) \wedge \\
&\qquad \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\}) \\
\underset{\substack{(4.20)\\(4.18)}}{\subseteq}\ &\{\widehat{t^{C_i}} \in \widehat{\mathit{ExecRuns}}^{C_i \prime\prime}_{prog,C_i} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}}^{LB}_{C_i}(\widehat{t^{C_i}}, x) \leq 0\} \\
\underset{(A.142)}{=}\ &\widehat{\mathit{Init}^{C_i}}
\end{aligned} \tag{A.151}$$

The induction start (i.e. $\mathit{IH}(0)$) is a direct consequence of statement (A.151).

$$
\begin{aligned}
& \pi^{C_i}(\widehat{\mathit{Cmpnd}''_0}) \underset{\text{(A.151)}}{\subseteq} \widehat{\mathit{Init}^{C_i}} \\
\underset{\substack{(4.31)\\(4.20)\\(4.18)}}{\Leftrightarrow}\ & \pi^{C_i}(\widehat{\mathit{Cmpnd}''_0}) \subseteq \pi^{C_i}(\chi(\overrightarrow{\mathit{Init}})) \\
\Leftrightarrow\ & [\forall \widehat{t^{C_i}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''_0}) : \widehat{t^{C_i}} \in \pi^{C_i}(\chi(\overrightarrow{\mathit{Init}}))] \\
\underset{(4.41)}{\Leftrightarrow}\ & [\forall \widehat{t^{C_i}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''_0}) : \\
& \quad \widehat{t^{C_i}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, ())))] \\
\underset{(4.40)}{\Rightarrow}\ & [\forall \widehat{t^{C_i}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''_0}) : \\
& \quad \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_i}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))] \\
\underset{\text{(A.150)}}{\Leftrightarrow}\ & \mathit{IH}(0)
\end{aligned}
$$

For the *induction step* (i.e. $\mathit{IH}(n) \Rightarrow \mathit{IH}(n+1)$), first, we define the set of members of $\pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}})$ which are not a member of $\pi^{C_i}(\widehat{\mathit{Cmpnd}''_n})$.

$$
\begin{aligned}
& \pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}}) \setminus \pi^{C_i}(\widehat{\mathit{Cmpnd}''_n}) \\
\underset{\text{(A.148)}}{=}\ & \pi^{C_i}(\{(\ldots, \widehat{t^{C_i}}, \ldots) \in \widehat{\mathit{Cmpnd}''} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}^{LB}_{C_i}}(\widehat{t^{C_i}}, x) \leq n+1\}) \setminus \\
& \pi^{C_i}(\{(\ldots, \widehat{t^{C_i}}, \ldots) \in \widehat{\mathit{Cmpnd}''} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}^{LB}_{C_i}}(\widehat{t^{C_i}}, x) \leq n\}) \\
=\ & \pi^{C_i}(\{(\ldots, \widehat{t^{C_i}}, \ldots) \in \widehat{\mathit{Cmpnd}''} \mid \sum_{x \in \mathbb{N}_{<len(\widehat{t^{C_i}})-1}} \widehat{\mathit{Blocked}^{LB}_{C_i}}(\widehat{t^{C_i}}, x) = n+1\})
\end{aligned}
\tag{A.152}
$$

Intuitively, for the members of $\pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}})$ which are also a member of $\pi^{C_i}(\widehat{\mathit{Cmpnd}''_n})$, $\mathit{IH}(n+1)$ holds as an immediate consequence of $\mathit{IH}(n)$. As a consequence, it is sufficient to only prove $\mathit{IH}(n+1)$ for the members of $\pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}}) \setminus \pi^{C_i}(\widehat{\mathit{Cmpnd}''_n})$. Thus, in the remainder of the induction step, we prove the following statement.

$$
\begin{gathered}
\forall \widehat{t^{C_i}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}}) \setminus \pi^{C_i}(\widehat{\mathit{Cmpnd}''_n}) : \\
\exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_i}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))
\end{gathered}
\tag{A.153}
$$

In the following, let $\widehat{t^{C_i *}}$ be an arbitrary member of $\pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}}) \setminus \pi^{C_i}(\widehat{\mathit{Cmpnd}''_n})$.

$$
\widehat{t^{C_i *}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''_{n+1}}) \setminus \pi^{C_i}(\widehat{\mathit{Cmpnd}''_n}) \tag{A.154}
$$

Based on the available information about $\widehat{t^{C_i*}}$, we can derive the following chain of implications.

$$\underset{\substack{(A.154)\\(A.152)}}{\Rightarrow} \quad [\exists(\ldots,\widehat{t^{C_{i-1}*}},\widehat{t^{C_i*}},\widehat{t^{C_{i+1}*}},\ldots)\in\widehat{Cmpnd}'':$$

$$\sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_i*}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i*}},x)=n+1]$$

$$\underset{\substack{(7.26)\\(7.22)\\(7.20)}}{\Leftrightarrow} \quad [\exists(\ldots,\widehat{t^{C_{i-1}*}},\widehat{t^{C_i*}},\widehat{t^{C_{i+1}*}},\ldots)\in\widehat{ExecRuns}^{C_1\prime\prime}_{prog,C_i}\times\widehat{ExecRuns}^{C_2\prime\prime}_{prog,C_i}\times\ldots:$$

$$(\exists m\in\mathbb{N}:\forall C_k\in Cores: len(\widehat{t^{C_k*}})=m)\wedge\widehat{P_{wc}}((\ldots,\widehat{t^{C_{i-1}*}},\widehat{t^{C_i*}},\widehat{t^{C_{i+1}*}},\ldots))\wedge$$

$$\sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_i*}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i*}},x)=n+1]$$

$$\underset{\substack{(7.33)\\(7.25)}}{\Rightarrow} \quad [\exists(\ldots,\widehat{t^{C_{i-1}**}},\widehat{t^{C_i**}},\widehat{t^{C_{i+1}**}},\ldots)\in\widehat{ExecRuns}^{C_1\prime\prime}_{prog,C_i}\times\widehat{ExecRuns}^{C_2\prime\prime}_{prog,C_i}\times\ldots:$$

$$(\exists m\in\mathbb{N}:\forall C_k\in Cores: len(\widehat{t^{C_k**}})=m)\wedge\widehat{P_{wc}}((\ldots,\widehat{t^{C_{i-1}**}},\widehat{t^{C_i**}},\widehat{t^{C_{i+1}**}},\ldots))\wedge$$

$$\sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_i**}})-1}} \widehat{Blocked}^{LB}_{C_i}(\widehat{t^{C_i**}},x)\leq n\wedge$$

$$(\widehat{t^{C_i**}},\widehat{t^{C_i*}})\in\widehat{PrefixOf}\wedge$$

$$n+1\leq\sum_{C_j\in Cores\setminus\{C_i\}}\ \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_j**}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j**}},x)]$$

$$\underset{\substack{(7.20)\\(7.22)\\(7.26)\\(A.148)}}{\Leftrightarrow} \quad [\exists(\ldots,\widehat{t^{C_{i-1}**}},\widehat{t^{C_i**}},\widehat{t^{C_{i+1}**}},\ldots)\in\widehat{Cmpnd}''_n:$$

$$(\widehat{t^{C_i**}},\widehat{t^{C_i*}})\in\widehat{PrefixOf}\wedge$$

$$\forall C_j\in Cores\setminus\{C_i\}: len(\widehat{t^{C_j**}})=len(\widehat{t^{C_i**}})\wedge$$

$$n+1\leq\sum_{C_j\in Cores\setminus\{C_i\}}\ \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_j**}})}} \widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j**}},x)]$$

The final statement of this chain of implications enables us to follow that there is a compound abstract trace $(\ldots,\widehat{t^{C_{i-1}**}},\widehat{t^{C_i**}},\widehat{t^{C_{i+1}**}},\ldots)\in\widehat{Cmpnd}''_n$ such that the following three statements are fulfilled.

$$\widehat{t^{C_i**}}\in\pi^{C_i}(\widehat{Cmpnd}''_n) \tag{A.155}$$

$$\forall C_j\in Cores\setminus\{C_i\}:\sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_j**}})}}\widehat{Cycle}^{LB}(\widehat{t^{C_j**}},x)\leq\sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_i**}})}}\widehat{Cycle}^{UB}(\widehat{t^{C_i**}},x) \tag{A.156}$$

$$n+1\leq\sum_{C_j\in Cores\setminus\{C_i\}}\ \sum_{x\in\mathbb{N}_{<len(\widehat{t^{C_j**}})}}\widehat{Granted}^{UB}_{C_j}(\widehat{t^{C_j**}},x) \tag{A.157}$$

With this additional machinery in place, we can conduct the actual induction step. It follows from *IH*(n) and statement (A.155) that there is a sequence of updates of approximation variables which adds $\widehat{t^{C_i**}}$ to the approximation variable of core C_i when applied to the vector of initial values of the approximation variables.

$$\begin{aligned}&\exists \mathit{updSequ} \in \mathit{UpdateSequ} : \exists \overrightarrow{\mathit{Approx}} :\\ &\overrightarrow{\mathit{Approx}} = \mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ}) \wedge \widehat{t^{C_i**}} \in \widehat{\mathit{Approx}}^{C_i}\end{aligned} \tag{A.158}$$

Next, based on this vector $\overrightarrow{\mathit{Approx}}$, we perform a simultaneous update of the approximation variables of the concurrent cores of core C_i.

$$\overrightarrow{\mathit{Approx}'} = \mathit{apply}(\overrightarrow{\mathit{Approx}}, (\mathit{Cores} \setminus \{C_i\})) \tag{A.159}$$

It follows from equations (A.156), (A.158), (A.159) and (A.133) that the approximation variable of each concurrent core C_j in the resulting vector contains the respective abstract trace $\widehat{t^{C_j**}}$.

$$\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \widehat{t^{C_j**}} \in \widehat{\mathit{Approx}'}^{C_j} \tag{A.160}$$

Thus, as a consequence of equations (A.157) and (A.160), the following statement is fulfilled.

$$n + 1 \leq \sum_{C_j \in \mathit{Cores} \setminus \{C_i\}} \max_{\widehat{t^{C_j}} \in \widehat{\mathit{Approx}'}^{C_j}} \sum_{\substack{x \in \mathbb{N} \\ < \mathit{len}(\widehat{t^{C_j}})}} \widehat{\mathit{Granted}}^{UB}_{C_j}(\widehat{t^{C_j}}, x) \tag{A.161}$$

Next, based on the vector $\overrightarrow{\mathit{Approx}'}$, we perform an update of the approximation variable of core C_i.

$$\overrightarrow{\mathit{Approx}''} = \mathit{apply}(\overrightarrow{\mathit{Approx}'}, (\{C_i\})) \tag{A.162}$$

As a consequence of equations (A.154), (A.152), (A.161), (A.162), and (A.141), the approximation variable of core C_i in the resulting vector contains abstract trace $\widehat{t^{C_i*}}$.

$$\widehat{t^{C_i*}} \in \widehat{\mathit{Approx}''}^{C_i} \tag{A.163}$$

This means that there is also an update sequence which creates $\widehat{t^{C_i*}}$ starting from the optimistic initialization.

$$\begin{aligned}&\widehat{t^{C_i*}} \underset{\text{(A.163)}}{\in} \widehat{\mathit{Approx}''}^{C_i}\\
\underset{\substack{(4.31)\\(4.20)\\(4.18)}}{\Leftrightarrow}\ & \widehat{t^{C_i*}} \in \pi^{C_i}(\chi(\overrightarrow{\mathit{Approx}''}))\\
\underset{\text{(A.162)}}{\Rightarrow}\ & \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_i*}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Approx}'}, \mathit{updSequ})))\\
\underset{\text{(A.159)}}{\Rightarrow}\ & \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_i*}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Approx}}, \mathit{updSequ})))\\
\underset{\text{(A.158)}}{\Rightarrow}\ & \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_i*}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))\end{aligned}$$

This concludes the proof of *IH*($n + 1$) and, thus, also the induction step.

As a consequence, we have proven statement (A.147). This means that we are only left to prove the part of statement (A.146) which argues about the concurrent cores C_j of core C_i.

$$\begin{aligned}&\forall C_j \in \mathit{Cores} \setminus \{C_i\} : \forall \widehat{t^{C_j}} \in \pi^{C_j}(\widehat{\mathit{Cmpnd}''}) :\\ &\quad \exists \mathit{updSequ} \in \mathit{UpdateSequ} : \widehat{t^{C_j}} \in \pi^{C_j}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))\end{aligned} \tag{A.164}$$

In the following, let $\widehat{t^{C_j*}}$ be an arbitrary member of $\pi^{C_j}(\widehat{\mathit{Cmpnd}''})$ for any core $C_j \neq C_i$.

$$\widehat{t^{C_j*}} \in \pi^{C_j}(\widehat{\mathit{Cmpnd}''}) \tag{A.165}$$

$$C_j \in \mathit{Cores} \setminus \{C_i\} \tag{A.166}$$

This time, we can reuse statement (A.147) in order to demonstrate that there has to be an update sequence which creates $\widehat{t^{C_j*}}$ starting from the optimistic initialization. In this way, we prove statement (A.164), which concludes the overall proof.

$$\underset{\substack{(A.165)\\(A.166)\\(7.26)\\(7.22)}}{\Rightarrow} [\exists \widehat{t^{C_i*}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''}) :$$

$$\mathit{len}(\widehat{t^{C_j*}}) \leq \mathit{len}(\widehat{t^{C_i*}})]$$

$$\underset{(5.40)}{\Rightarrow} [\exists \widehat{t^{C_i*}} \in \pi^{C_i}(\widehat{\mathit{Cmpnd}''}) :$$

$$\sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_j*}})}} \widehat{\mathit{Cycle}^{LB}}(\widehat{t^{C_j*}}, x) \leq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_i*}})}} \widehat{\mathit{Cycle}^{UB}}(\widehat{t^{C_i*}}, x)]$$

$$\underset{(A.147)}{\Rightarrow} [\exists \mathit{updSequ} \in \mathit{UpdateSequ} :$$

$$\exists \widehat{t^{C_i*}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ}))) :$$

$$\sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_j*}})}} \widehat{\mathit{Cycle}^{LB}}(\widehat{t^{C_j*}}, x) \leq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_i*}})}} \widehat{\mathit{Cycle}^{UB}}(\widehat{t^{C_i*}}, x)]$$

$$\underset{\substack{(A.133)\\(A.166)}}{\Leftrightarrow} [\exists \mathit{updSequ} \in \mathit{UpdateSequ} :$$

$$\exists \widehat{t^{C_i*}} \in \pi^{C_i}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ}))) :$$

$$\sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_j*}})}} \widehat{\mathit{Cycle}^{LB}}(\widehat{t^{C_j*}}, x) \leq \sum_{x \in \mathbb{N}_{<\mathit{len}(\widehat{t^{C_i*}})}} \widehat{\mathit{Cycle}^{UB}}(\widehat{t^{C_i*}}, x) \wedge$$

$$\widehat{t^{C_j*}} \in \pi^{C_j}(\chi(\mathit{apply}(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ}), (\{C_j\}))))]$$

$$\Rightarrow [\exists \mathit{updSequ} \in \mathit{UpdateSequ} :$$

$$\widehat{t^{C_j*}} \in \pi^{C_j}(\chi(\mathit{apply}(\overrightarrow{\mathit{Init}}, \mathit{updSequ})))] \quad \square$$

Proof Sketch Demonstrating that the Calculation Procedure Presented in Section 8.2 Results in a Compositional Base Bound Whenever It Results in a Defined Value. The soundness statements of the calculation procedure presented in Section 8.2 are given by equations (8.7) and (8.8). Based on equations (8.5) and (8.6), we derive the following equivalent soundness statements.

$$\begin{aligned}
&\mathit{Maximum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} \in \mathbb{R} \\
&\Rightarrow \forall t \in \mathit{ExecRuns}_{prog,C_i} : \\
&\quad \mathit{numEvOccur}(prog, C_i, t, E_{bnd}) - \sum_{(pen,E)\in PenEv} pen \cdot \mathit{numEvOccur}(prog, C_i, t, E) \\
&\qquad \leq \mathit{Maximum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv}
\end{aligned} \tag{A.167}$$

$$\begin{aligned}
&\mathit{Minimum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv} \in \mathbb{R} \\
&\Rightarrow \forall t \in \mathit{ExecRuns}^{min\text{-}relev}_{prog,C_i,E} : \\
&\quad \mathit{numEvOccur}(prog, C_i, t, E_{bnd}) - \sum_{(pen,E)\in PenEv} pen \cdot \mathit{numEvOccur}(prog, C_i, t, E) \\
&\qquad \geq \mathit{Minimum}^{\widehat{B,impli}}_{prog,C_i,E_{bnd},PenEv}
\end{aligned} \tag{A.168}$$

Now, we sketch a proof of statements (A.167) and (A.168). The calculation procedure presented in Section 8.2 operates at the level of approximation of implicit path enumeration. In order to demonstrate its soundness (i.e. statements (A.167) and (A.168)), first, we specify the corresponding calculation at the level of approximation of paths through a graph $G^{B\prime}$.

$$\begin{aligned}
\mathit{Maximum}^{\widehat{B\prime,path}}_{prog,C_i,E_{bnd},PenEv} = &\max_{\widehat{p}\in \widehat{LessFeedPaths}^{B\prime}_{prog,C_i}} \sum_{x\in\mathbb{N}_{<len(\widehat{p})}} [\widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(\widehat{p}, x) \\
&- \sum_{(pen,E)\in PenEv} pen \cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(\widehat{p}, x)]
\end{aligned} \tag{A.169}$$

$$\begin{aligned}
\mathit{Minimum}^{\widehat{B\prime,path}}_{prog,C_i,E_{bnd},PenEv} = &\min_{\widehat{p}\in \widehat{LessFeedPaths}^{B\prime}_{prog,C_i}} \sum_{x\in\mathbb{N}_{<len(\widehat{p})}} [\widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(\widehat{p}, x) \\
&- \sum_{(pen,E)\in PenEv} pen \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(\widehat{p}, x)]
\end{aligned} \tag{A.170}$$

The following statements claim the soundness of these bounds calculated at the level of approximation of paths through a graph. Their proof is analogous to the proof of statements (6.129) and (6.130) on page 289 and, thus, omitted.

$$\begin{aligned}
&[G^{B\prime} \models G^{detail,prog,C_i} \wedge \mathit{Maximum}^{\widehat{B\prime,path}}_{prog,C_i,E_{bnd},PenEv} \in \mathbb{R}] \\
&\Rightarrow \forall t \in \mathit{ExecRuns}_{prog,C_i} : \\
&\quad \mathit{numEvOccur}(prog, C_i, t, E_{bnd}) - \sum_{(pen,E)\in PenEv} pen \cdot \mathit{numEvOccur}(prog, C_i, t, E) \\
&\qquad \leq \mathit{Maximum}^{\widehat{B\prime,path}}_{prog,C_i,E_{bnd},PenEv}
\end{aligned} \tag{A.171}$$

$$\begin{aligned}
&[G^{B\prime} \models G^{detail,prog,C_i} \land \widehat{Minimum}^{B\prime,path}_{prog,C_i,E_{bnd},PenEv} \in \mathbb{R}] \\
&\Rightarrow \forall t \in ExecRuns^{min\text{-}relev}_{prog,C_i,E} : \\
&\quad numEvOccur(prog, C_i, t, E_{bnd}) - \sum_{(pen,E)\in PenEv} pen \cdot numEvOccur(prog, C_i, t, E) \\
&\quad\quad \geq \widehat{Minimum}^{B\prime,path}_{prog,C_i,E_{bnd},PenEv}
\end{aligned} \tag{A.172}$$

Our graph construction algorithm (cf. Section 6.4.4) guarantees that the graph $G^{B\prime}$ it constructs is guaranteed to subsume the detailed graph—as required by statements (A.171) and (A.172). Before we perform the implicit path enumeration, we add a set of feedback nodes to the end nodes of the graph and optionally perform graph transformations (cf. our standard workflow as described in Figure 6.5). The result of these operations is the graph G^B.

Finally, statements (A.167) and (A.168) follow from statements (A.171) and (A.172), the soundness of potential graph transformations, the soundness of implicit path enumeration, and the soundness of property lifting. □

Proof that the Alternative Implementation Presented in Section 8.2 Provides the Same Results as the Calculation Procedure Presented in Section 8.2. First, we prove that the alternative implementation for calculating upper-bounding compositional base bounds (i.e. following equations (8.9) and (8.10)) leads to the same result as the calculation procedure following equation (8.5).

$$\begin{aligned}
&\widehat{Maximum}^{B,impli}_{prog,C_i,E_{bnd},PenEv} \\
\underset{(8.5)}{=}\; & \max_{(tt,*,*)\in \widehat{LessImplicit}^B} \sum_{edg\in Edges^B} tt(edg) \cdot \\
& \quad [\widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(edg) - \sum_{(pen,E)\in PenEv} pen \cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(edg)] \\
\underset{\substack{\text{distributivity}\\ \text{commutativity}}}{=}\; & \max_{(tt,*,*)\in \widehat{LessImplicit}^B} [\sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(edg) \\
& \quad - \sum_{(pen,E)\in PenEv} pen \cdot \sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{LB}_{prog,C_i,E}(edg)] \\
\underset{(8.9)}{=}\; & \max_{\substack{(tt,*,*)\in \widehat{LessImplicit}^B \\ \forall (pen,E)\in PenEv: var_{(pen,E)}\in\mathbb{Z}}} [\sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_i,E_{bnd}}(edg) \\
& \quad - \sum_{(pen,E)\in PenEv} pen \cdot var_{(pen,E)}]
\end{aligned}$$

According to Section 8.2, there is a corresponding alternative implementation for the calculation of lower-bounding compositional base bounds. Analogously to equations (8.9) and (8.10), it is given by the following equations.

$$\forall (pen, E) \in PenEv : \sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(edg) \geq var_{(pen,E)} \tag{A.173}$$

$$\begin{aligned}
&\min_{\substack{(tt,*,*)\in \widehat{LessImplicit}^B \\ \forall (pen,E)\in PenEv: var_{(pen,E)}\in\mathbb{Z}}} [\sum_{edg\in Edges^B} tt(edg) \cdot \widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(edg) \\
&\quad - \sum_{(pen,E)\in PenEv} pen \cdot var_{(pen,E)}]
\end{aligned} \tag{A.174}$$

Finally, we demonstrate that the calculation of lower-bounding compositional base bounds following equations (A.173) and (A.174) leads to the same result as the calculation procedure following equation (8.6).

$$
\begin{aligned}
& \widehat{Minimum}^{B,impli}_{prog,C_i,E_{bnd},PenEv} \\
\underset{(8.6)}{=} \quad & \min_{(tt,*,*)\in \widehat{LessImplicit}^{B}} \sum_{edg\in Edges^{B}} tt(edg) \cdot \\
& \quad [\widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(edg) - \sum_{(pen,E)\in PenEv} pen \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(edg)] \\
\underset{\substack{\text{distributivity}\\ \text{commutativity}}}{=} \quad & \min_{(tt,*,*)\in \widehat{LessImplicit}^{B}} [\sum_{edg\in Edges^{B}} tt(edg) \cdot \widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(edg) \\
& \quad - \sum_{(pen,E)\in PenEv} pen \cdot \sum_{edg\in Edges^{B}} tt(edg) \cdot \widehat{wEvent}^{UB}_{prog,C_i,E}(edg)] \\
\underset{(A.173)}{=} \quad & \min_{\substack{(tt,*,*)\in \widehat{LessImplicit}^{B}\\ \forall (pen,E)\in PenEv: var_{(pen,E)}\in \mathbb{Z}}} [\sum_{edg\in Edges^{B}} tt(edg) \cdot \widehat{wEvent}^{LB}_{prog,C_i,E_{bnd}}(edg) \\
& \quad - \sum_{(pen,E)\in PenEv} pen \cdot var_{(pen,E)}] \quad \square
\end{aligned}
$$

Proof Sketch that Equations (10.102) *to* (10.105) *Precisely Simulate the Equivalence in Equation* (10.101). Equation (10.101) is precisely simulated by the following set of implications.

$$
\begin{aligned}
& \forall prog \in Programs_{C_j} : \\
& \quad \sum_{prog' \in Programs_{C_j} \setminus \{prog\}} Runs_{prog'} > 0 \Rightarrow SingletonRun_{prog} = 0 \qquad \text{(A.175)}
\end{aligned}
$$

$$
\sum_{prog\in Programs_{C_j}} Runs_{prog} = 1 \Rightarrow \sum_{prog\in Programs_{C_j}} SingletonRun_{prog} = 1 \qquad \text{(A.176)}
$$

The intuition behind these implications is as follows. A program *prog* cannot perform a singleton run in case a run of one of the other programs is executed as well (cf. equation (A.175)). Moreover, in case there is only a single execution run over all programs, there has to be a program performing a singleton execution run (cf. equation (A.176)).

It follows from equations (10.96), (10.113), (10.114), (10.115), (10.116), and (10.117) that the interval length l is an upper bound of the sum of execution runs over all programs.

$$
\sum_{prog\in Programs_{C_j}} Runs_{prog} \leq l \qquad \text{(A.177)}
$$

Thus, we use l as a Big-M in equation (10.102) in order to precisely simulate equation (A.175).

The implication of equation (A.176), in contrast, is further split into the following three implications that argue about the binary helper variables helper_1 and helper_2.

$$\sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{Runs}_{\mathit{prog}} \geq 1 \Rightarrow \mathit{helper}_1 = 1 \tag{A.178}$$

$$\sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{Runs}_{\mathit{prog}} \leq 1 \Rightarrow \mathit{helper}_2 = 1 \tag{A.179}$$

$$\mathit{helper}_1 \wedge \mathit{helper}_2 \Rightarrow \sum_{\mathit{prog} \in \mathit{Programs}_{C_j}} \mathit{SingletonRun}_{\mathit{prog}} = 1 \tag{A.180}$$

Once again, we use l as a Big-M in equation (10.103) in order to precisely simulate equation (A.178). Moreover, equation (A.179) respectively (A.180) is precisely simulated by equation (10.104) respectively (10.105). □

Appendix B

Backup Experiment Results

– Where's the fucking money Keith?
– Dad calm down! Listen to me, the money is not important here...

(Some Kind of Wonderful, 1987)

This appendix chapter contains additional experiment results that have not been presented in the main part of this thesis for the sake of readability.

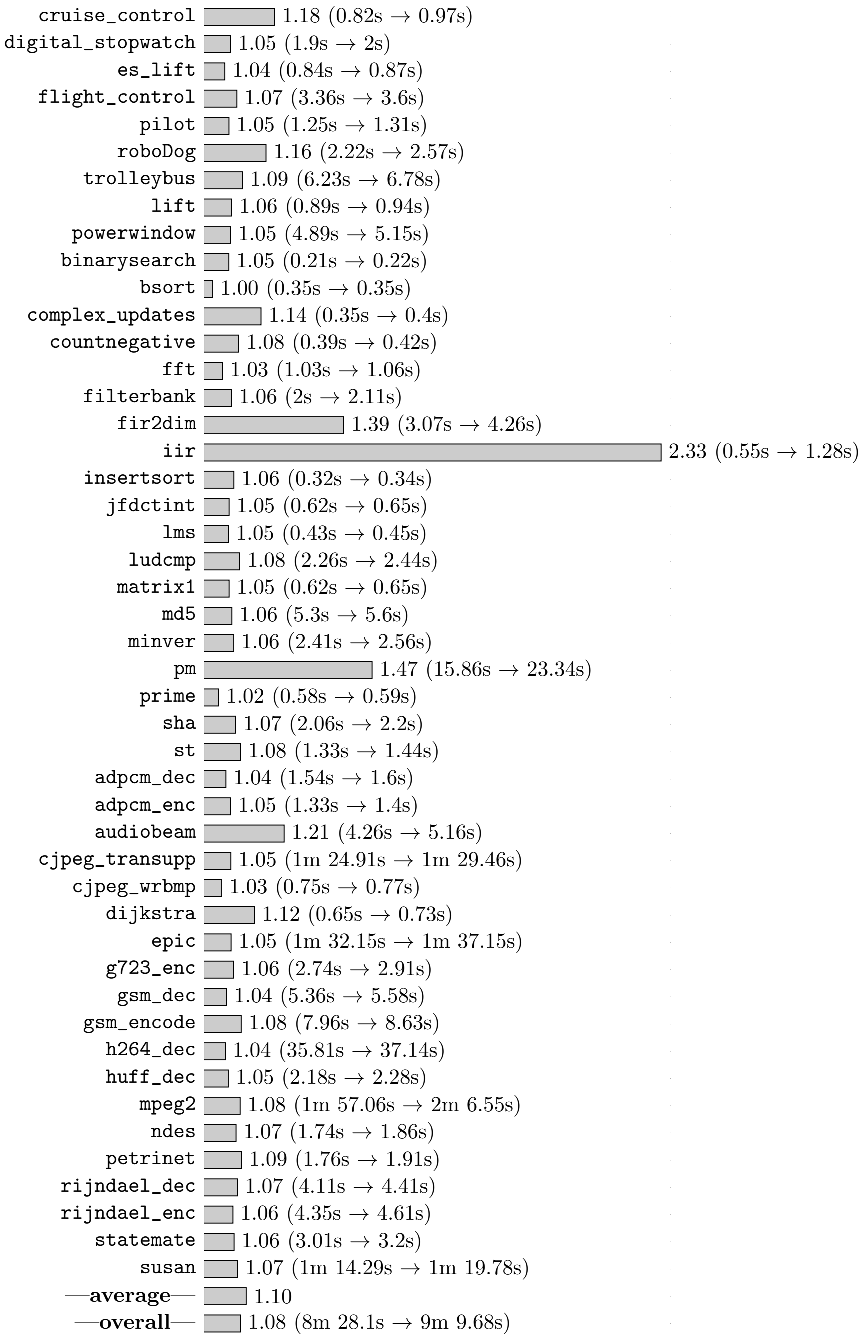


Figure B.1.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{is}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

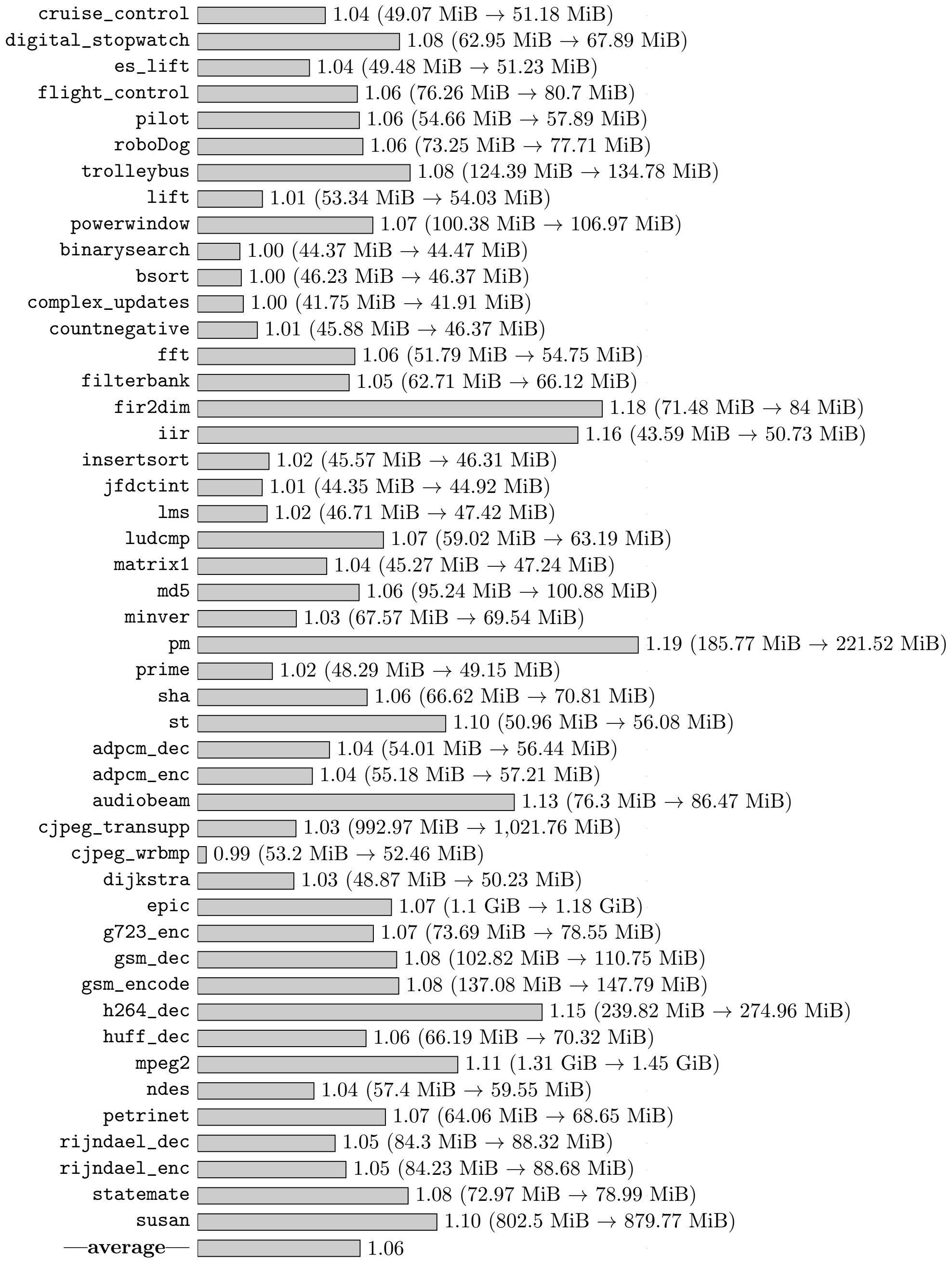


Figure B.2.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{is}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

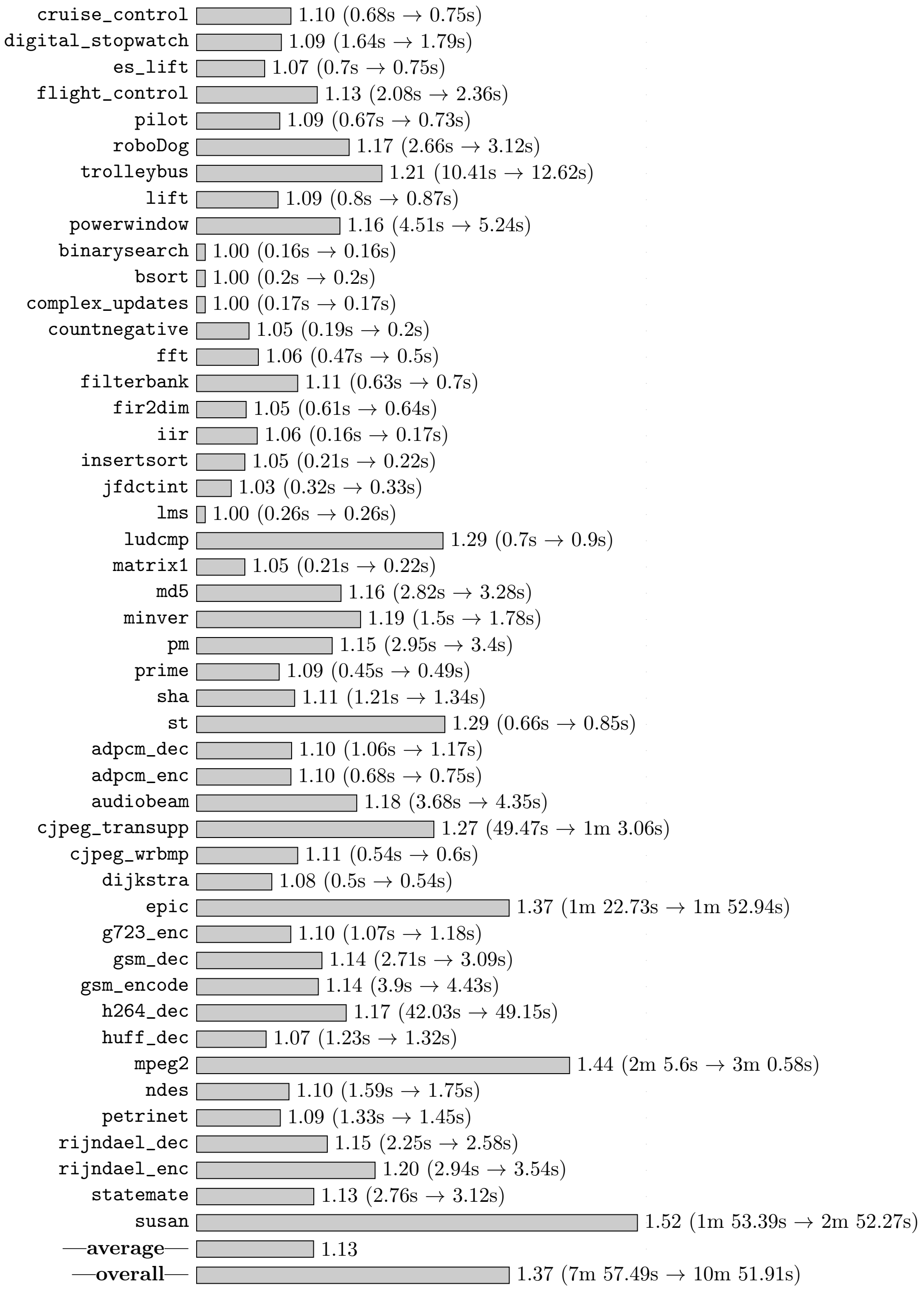


Figure B.3.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{ic}^{io}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

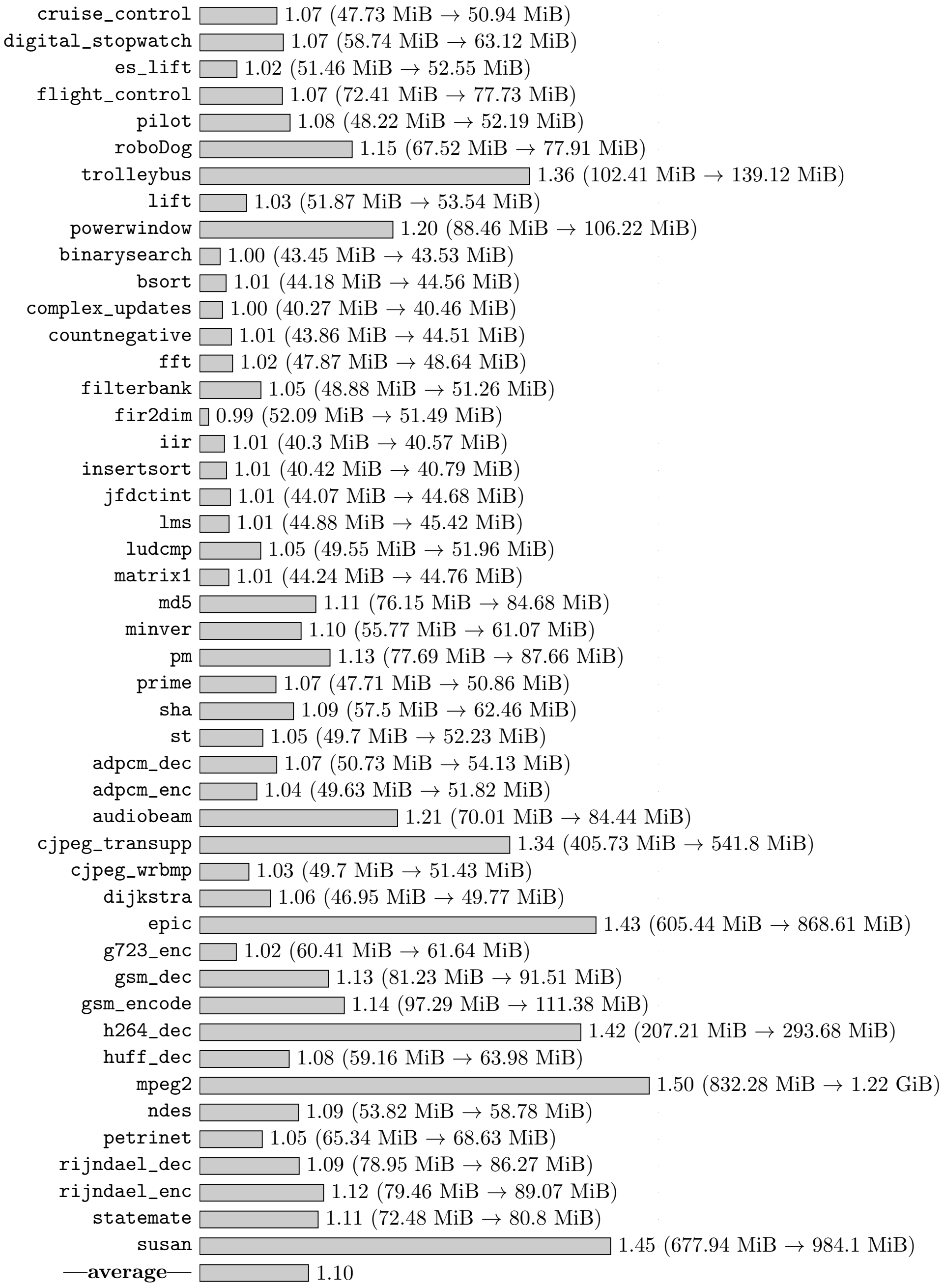


Figure B.4.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{ic}^{io}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

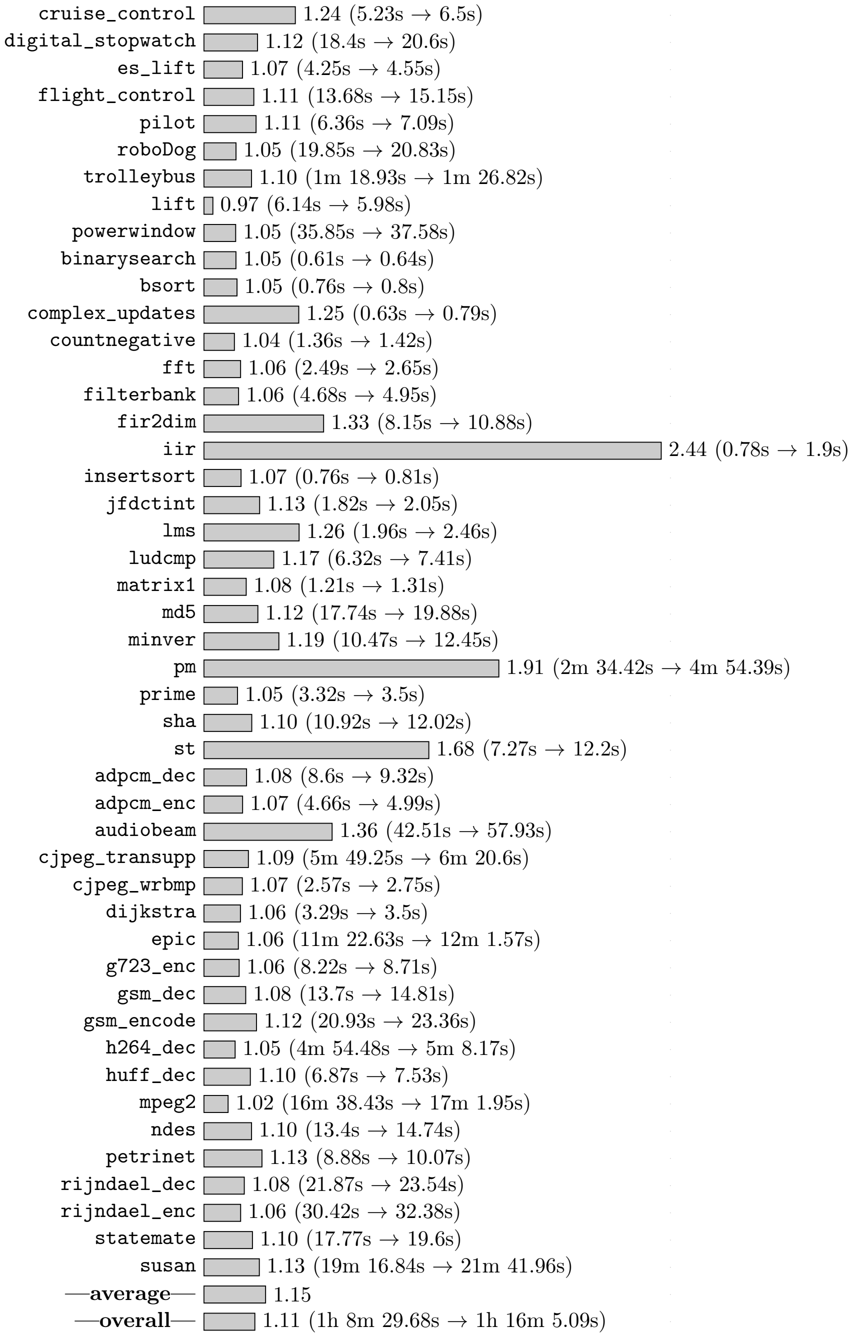


Figure B.5.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

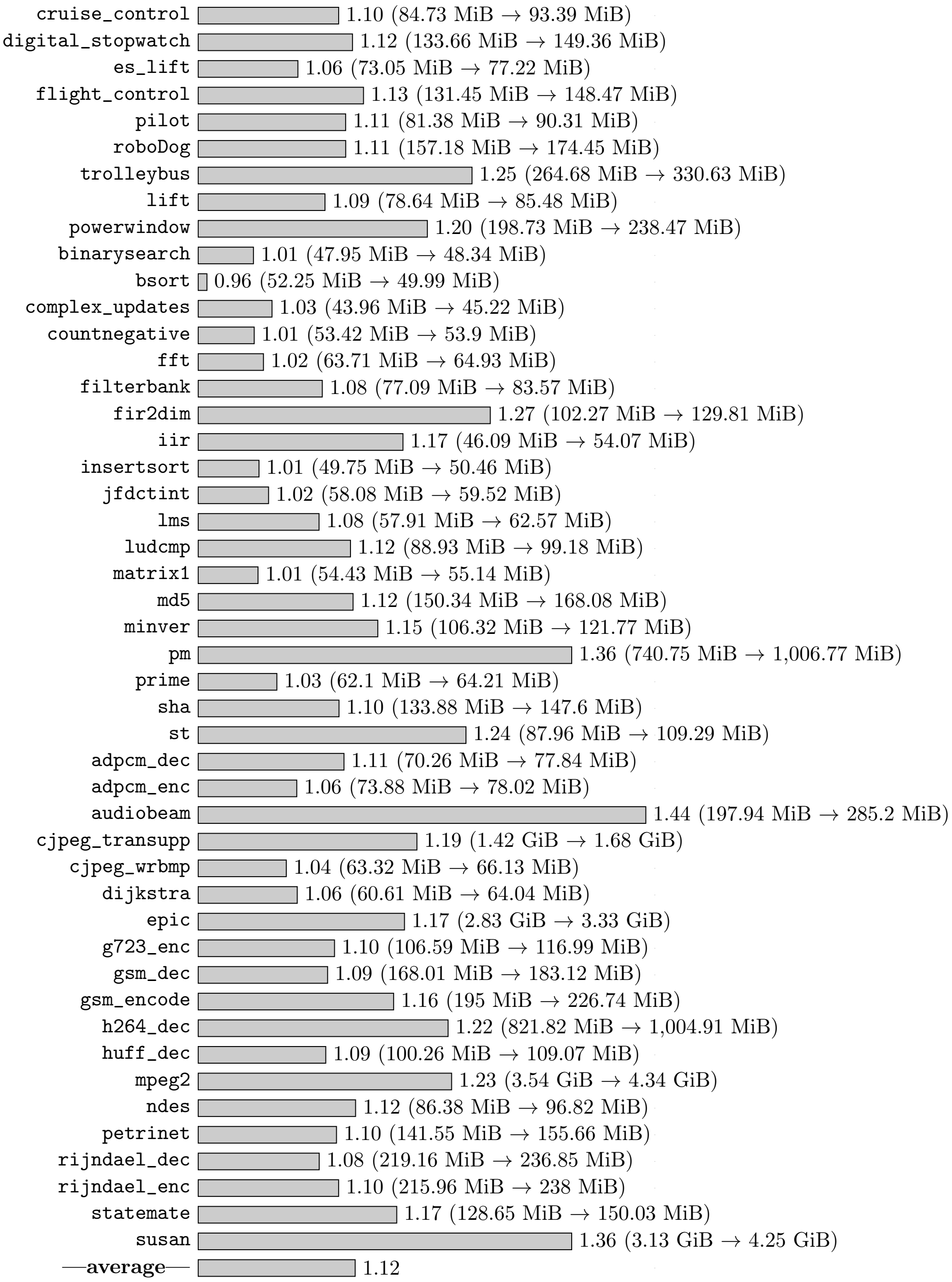


Figure B.6.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

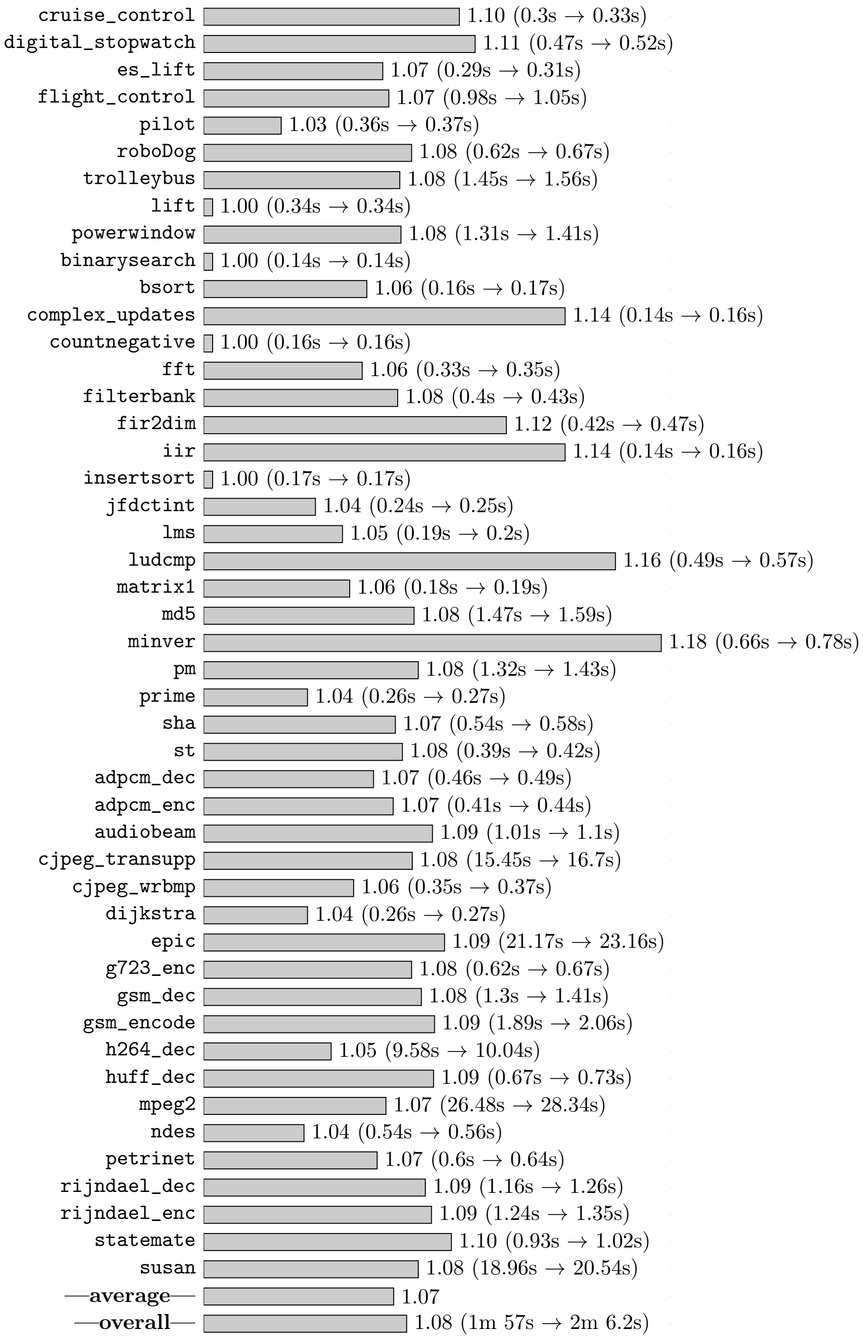


Figure B.7.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{is}^{io}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

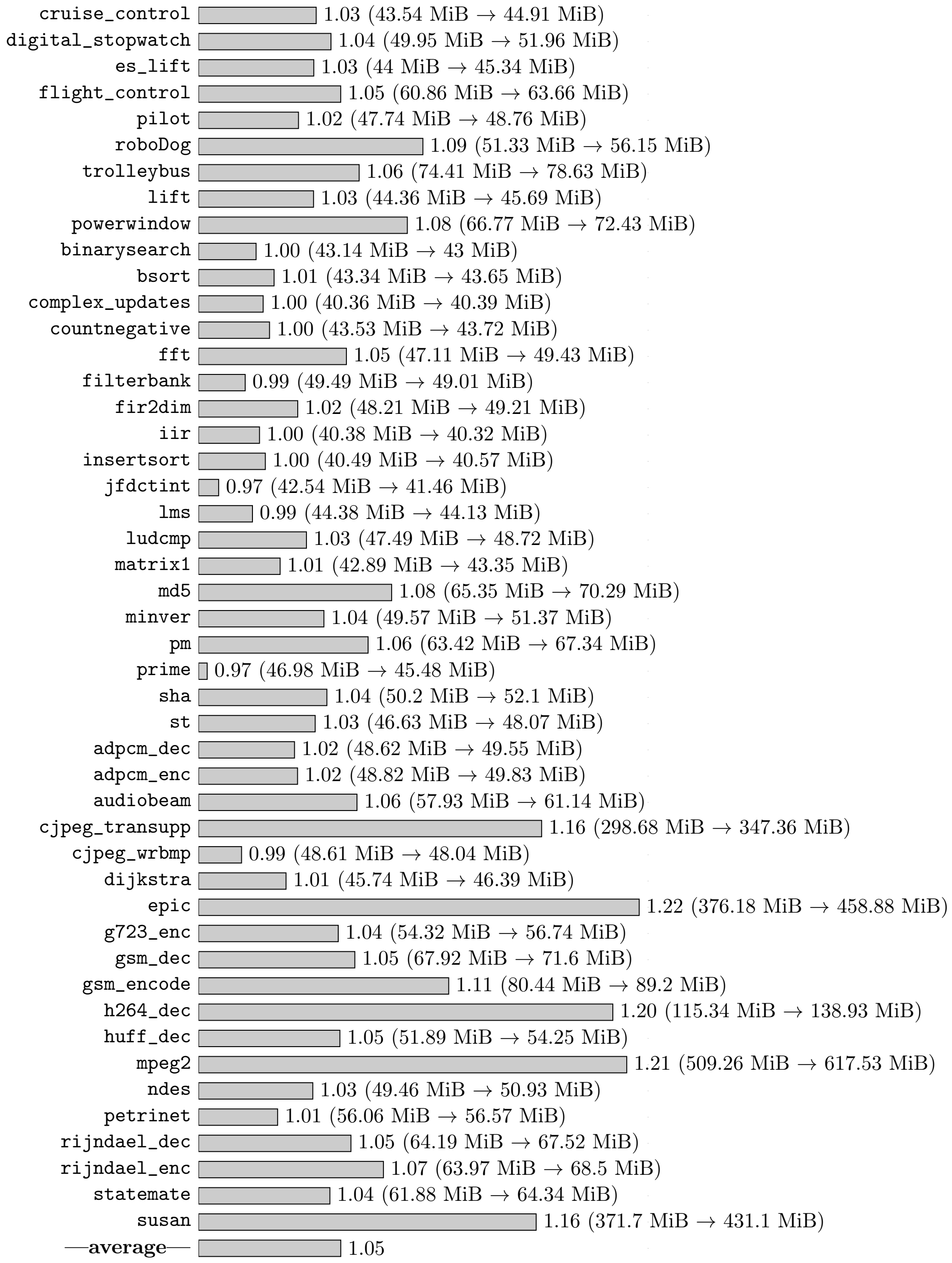


Figure B.8.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{is}^{io}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

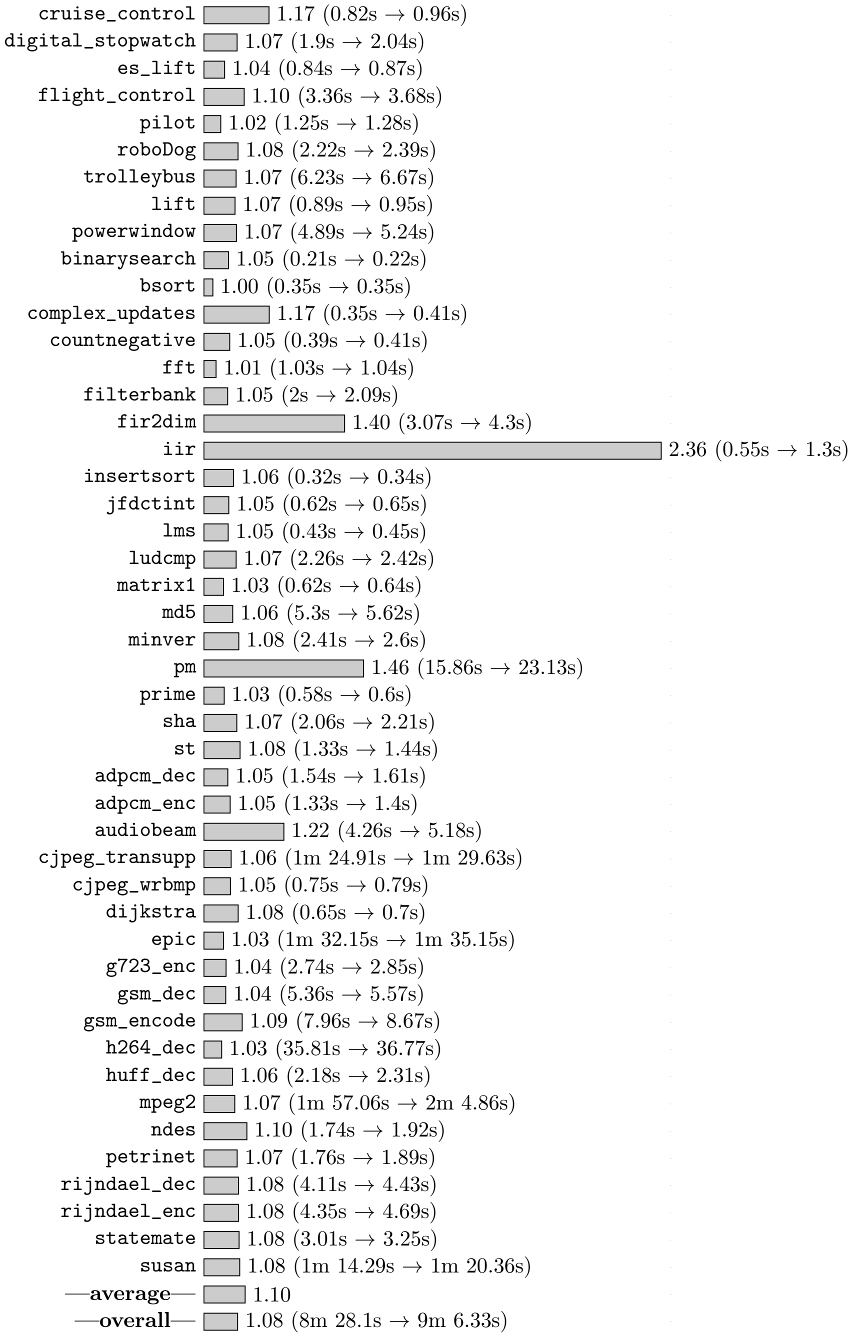


Figure B.9.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{is}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

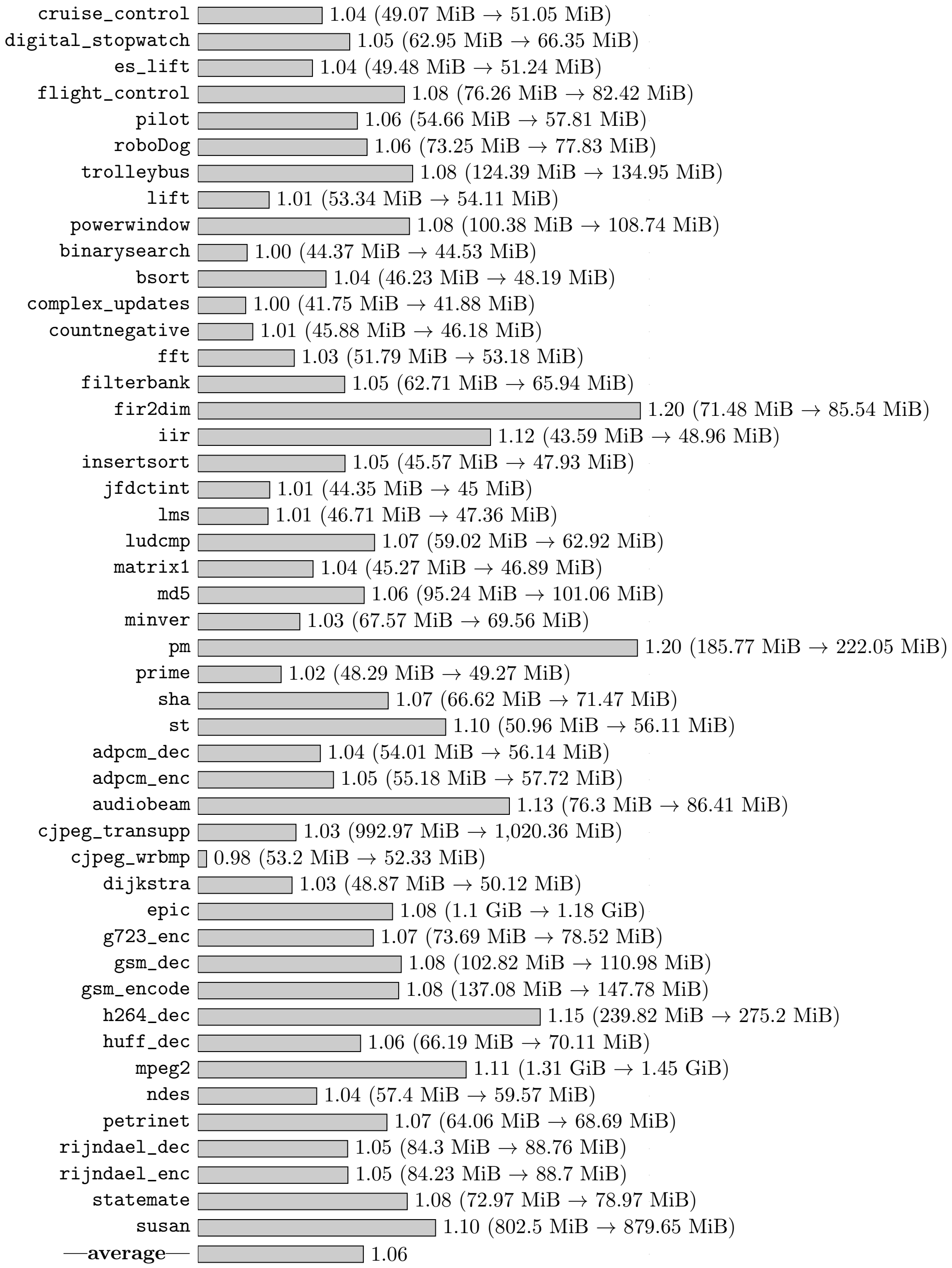


Figure B.10.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{is}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

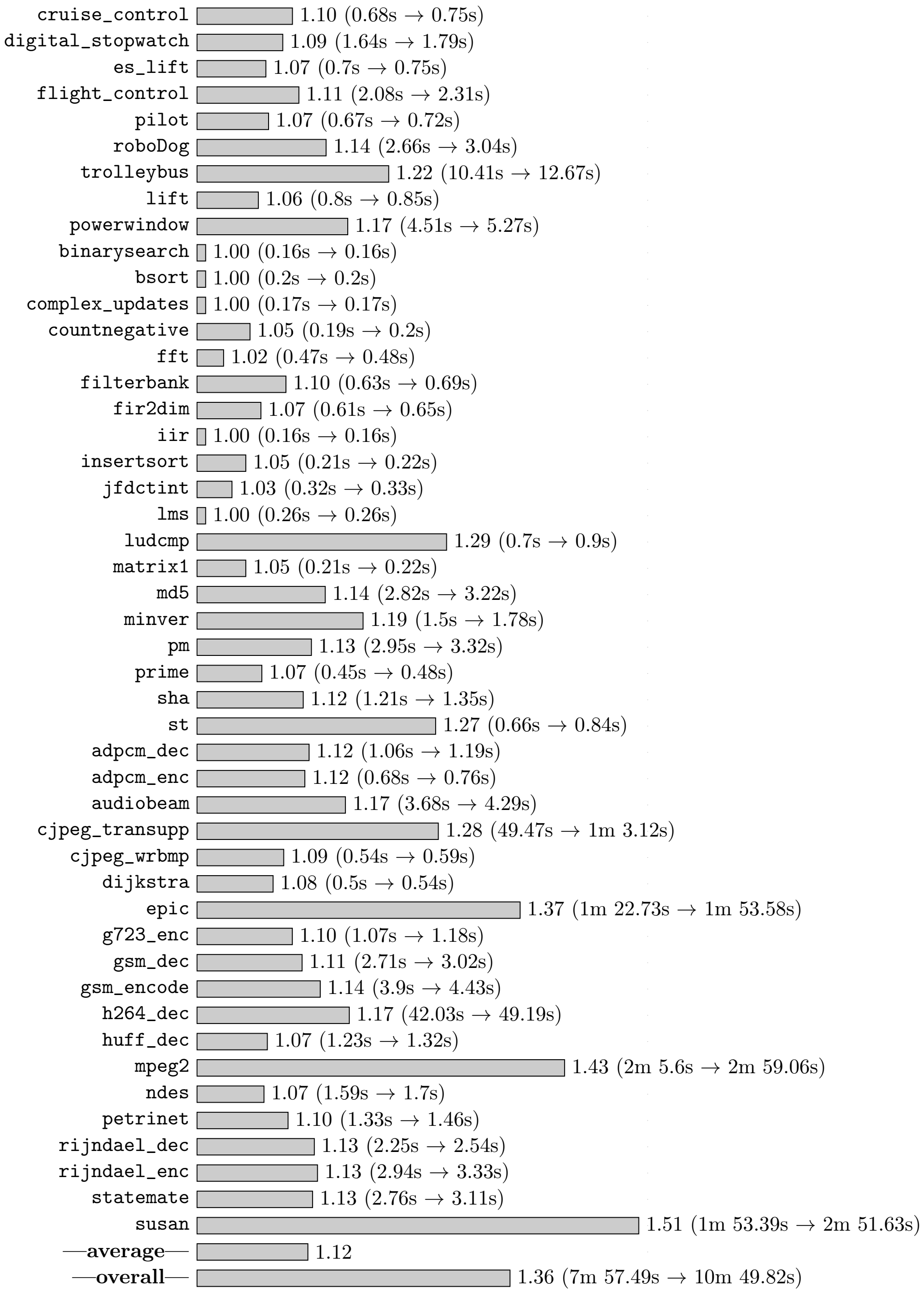


Figure B.11.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{ic}^{io}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

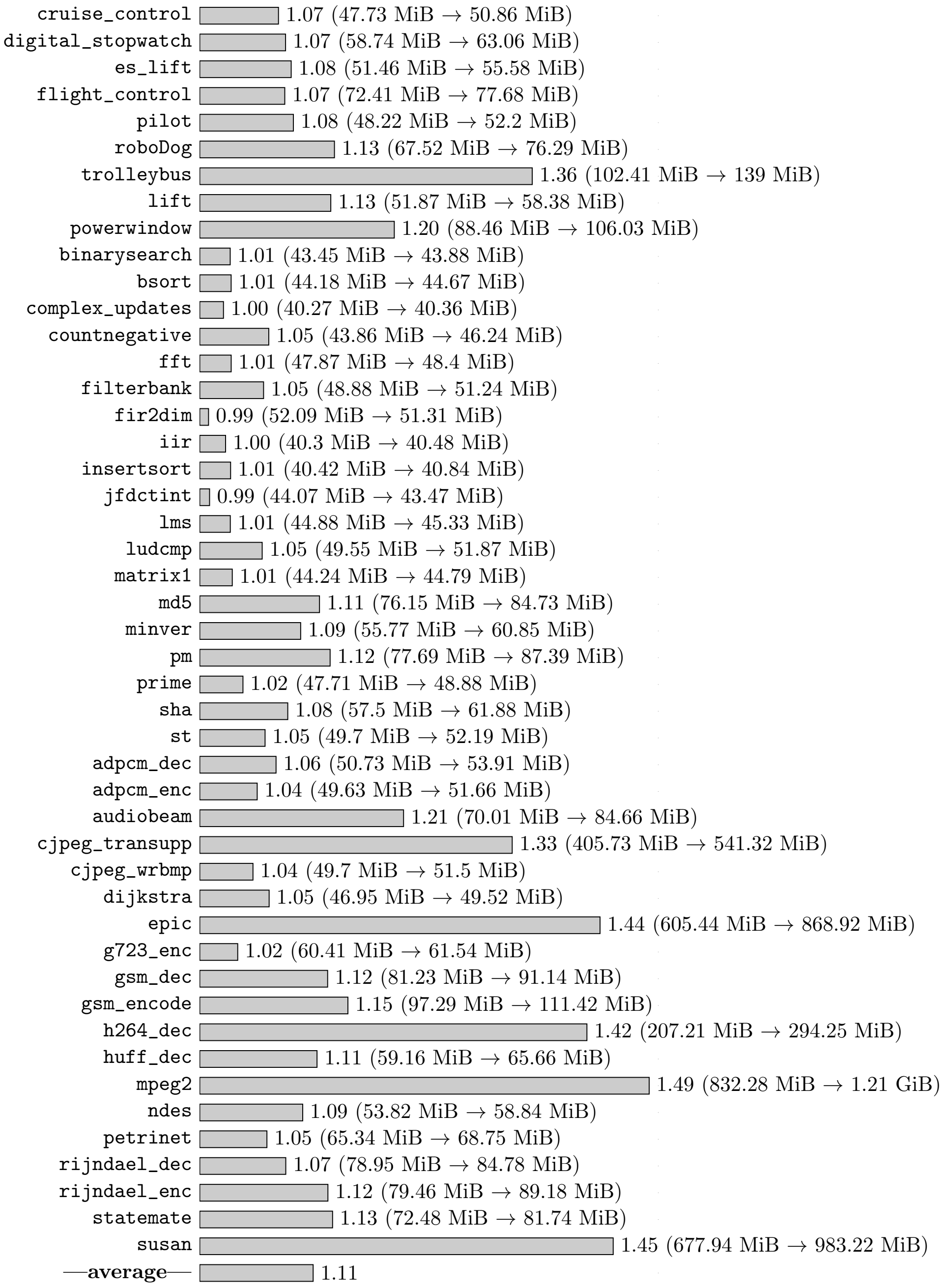


Figure B.12.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{ic}^{io}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

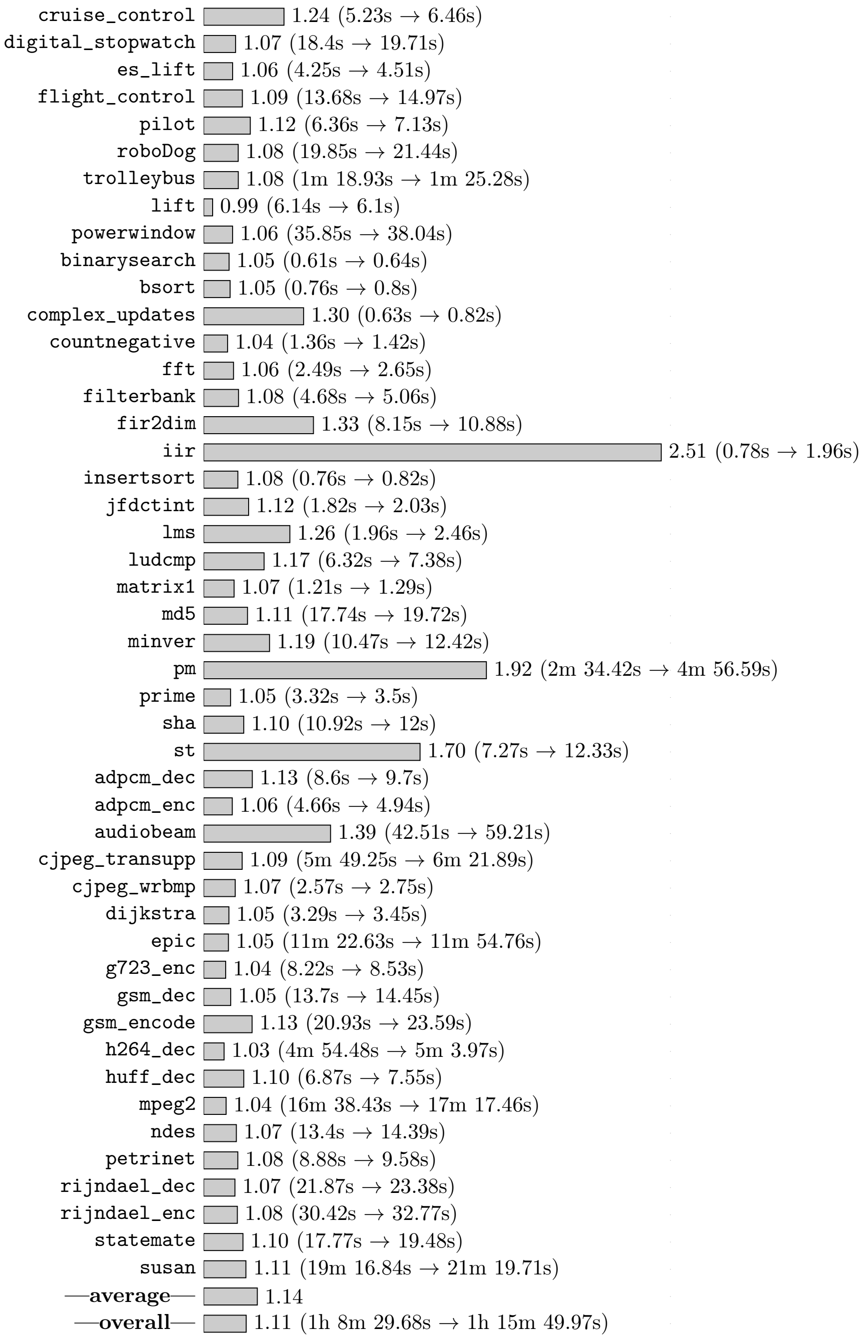


Figure B.13.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of an analysis assuming the absence of shared-bus interference.

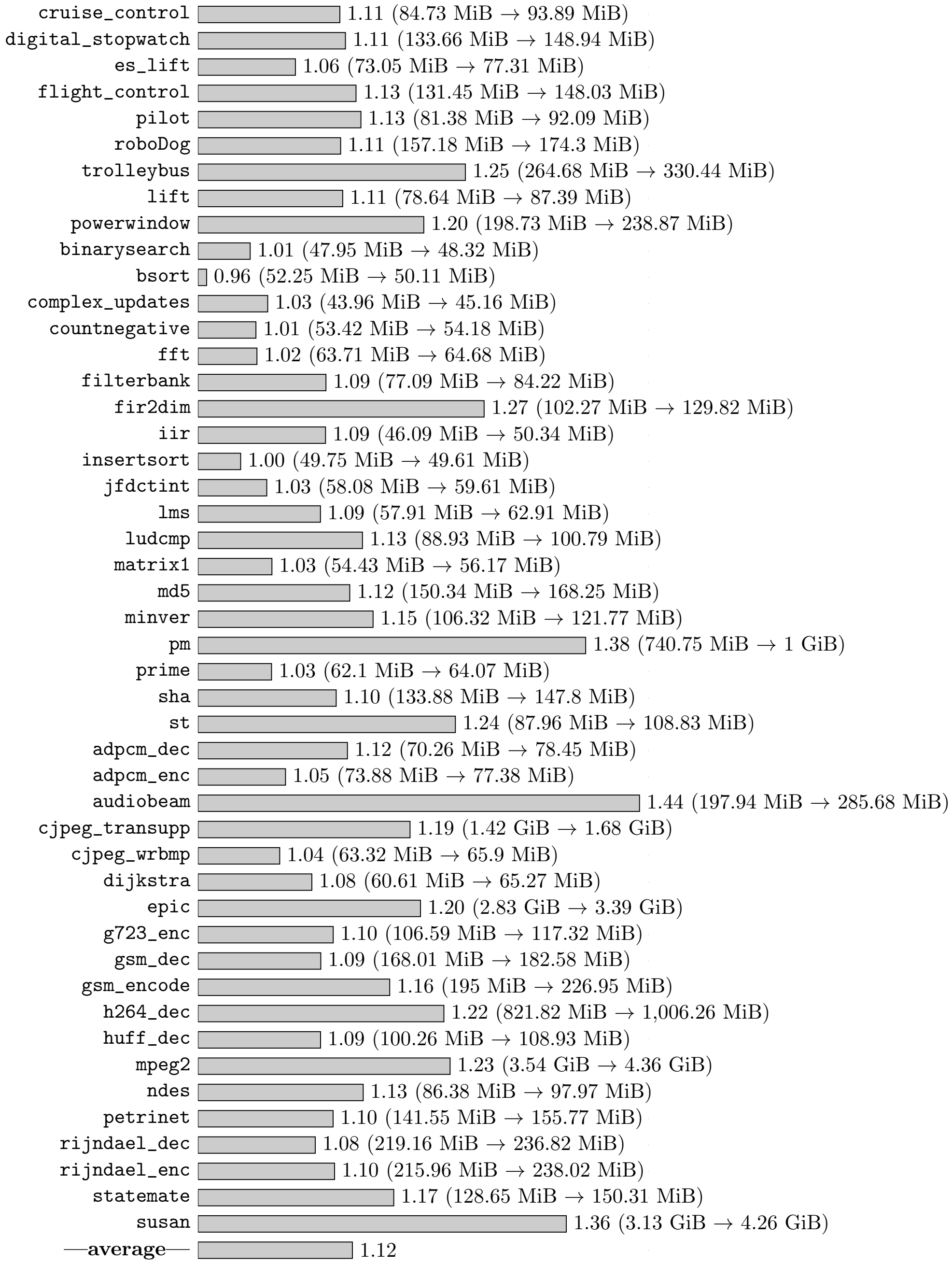


Figure B.14.: Co-runner-insensitive WCET analysis (with fast-forwarding of converged chains and delayed case splits) for an octa-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of an analysis assuming the absence of shared-bus interference.

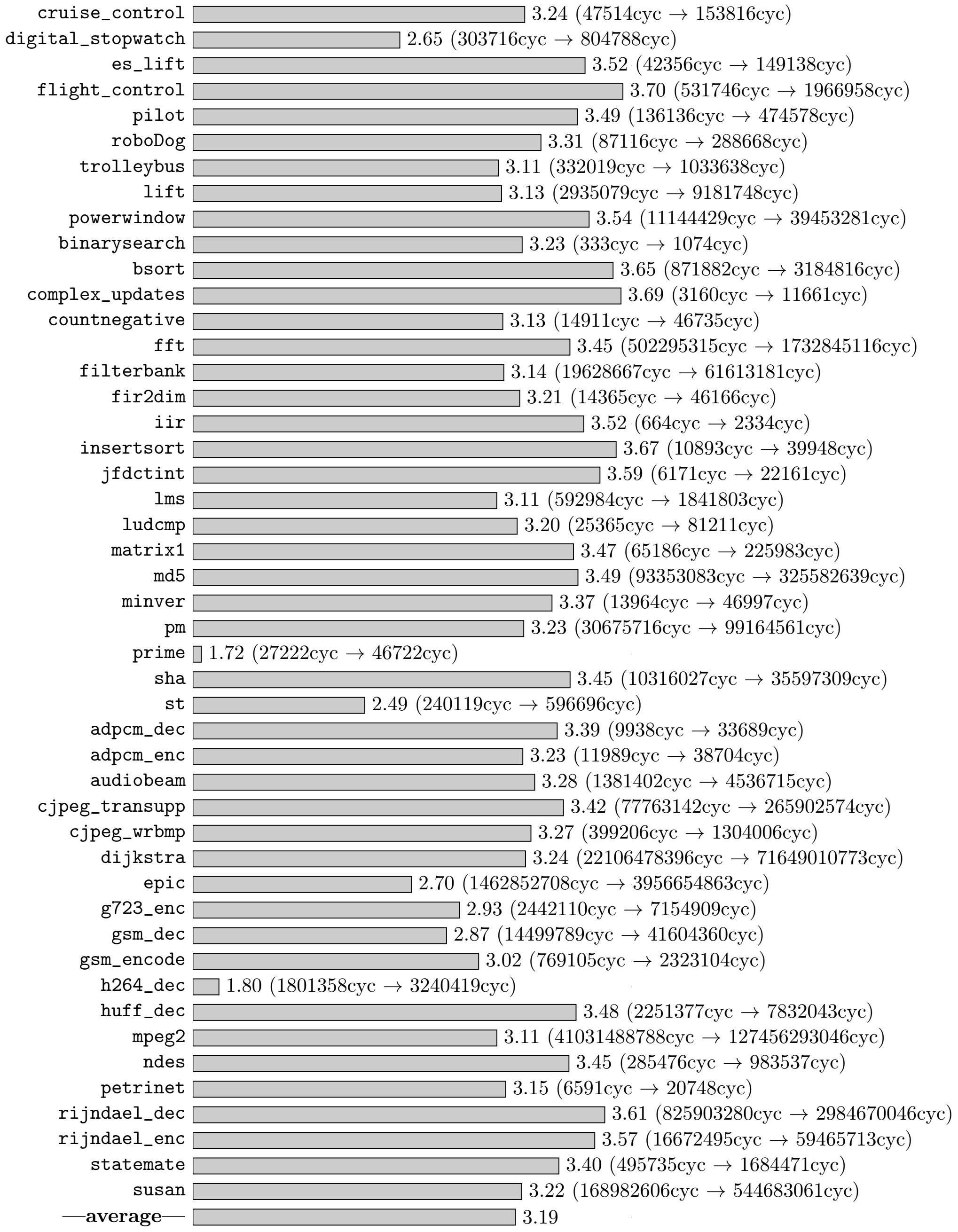


Figure B.15.: Co-runner-insensitive WCET analysis for a quad-core processor with core configuration $Conf_{is}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

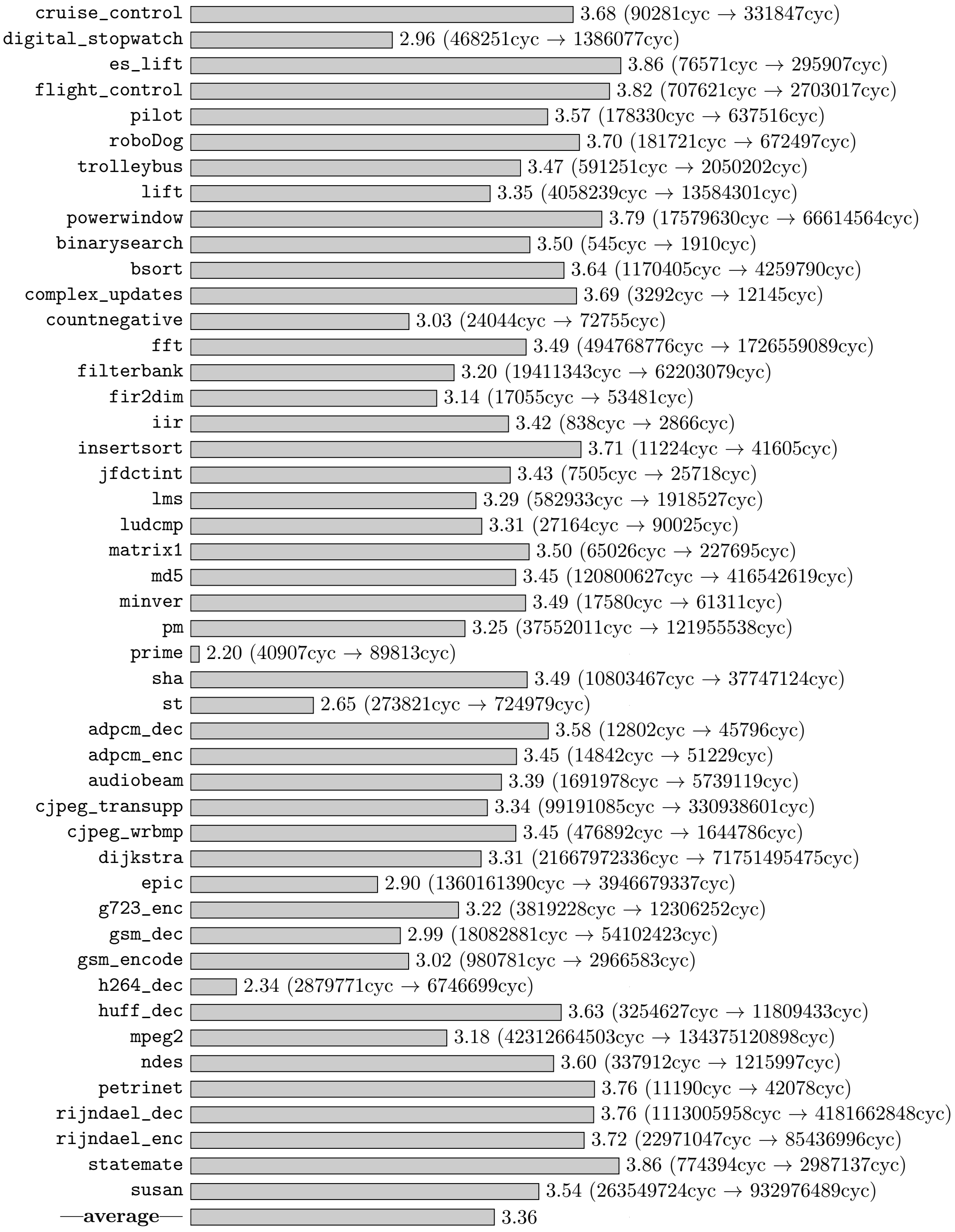


Figure B.16.: Co-runner-insensitive WCET analysis for a quad-core processor with core configuration $Conf_{ic}^{io}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

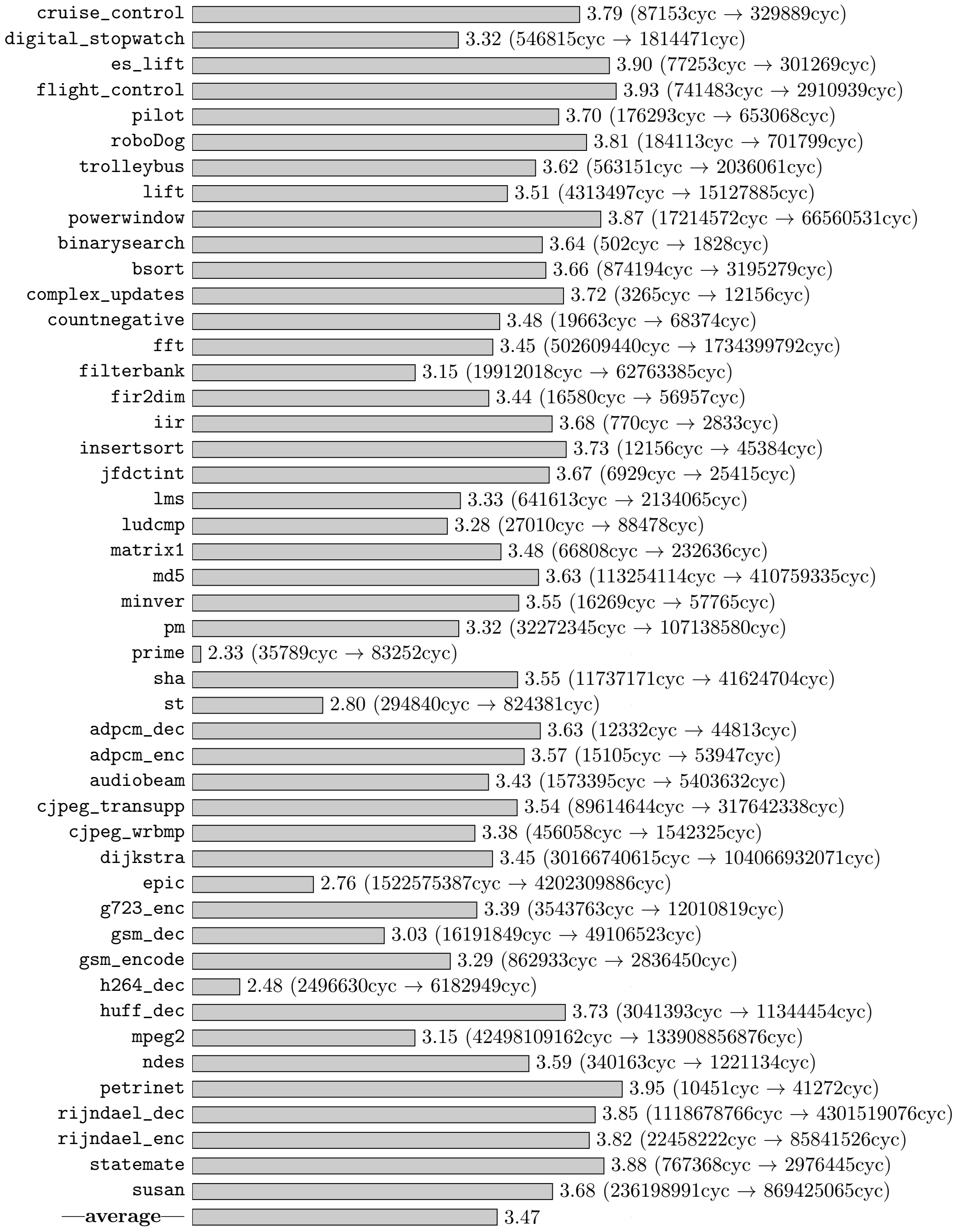


Figure B.17.: Co-runner-insensitive WCET analysis for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

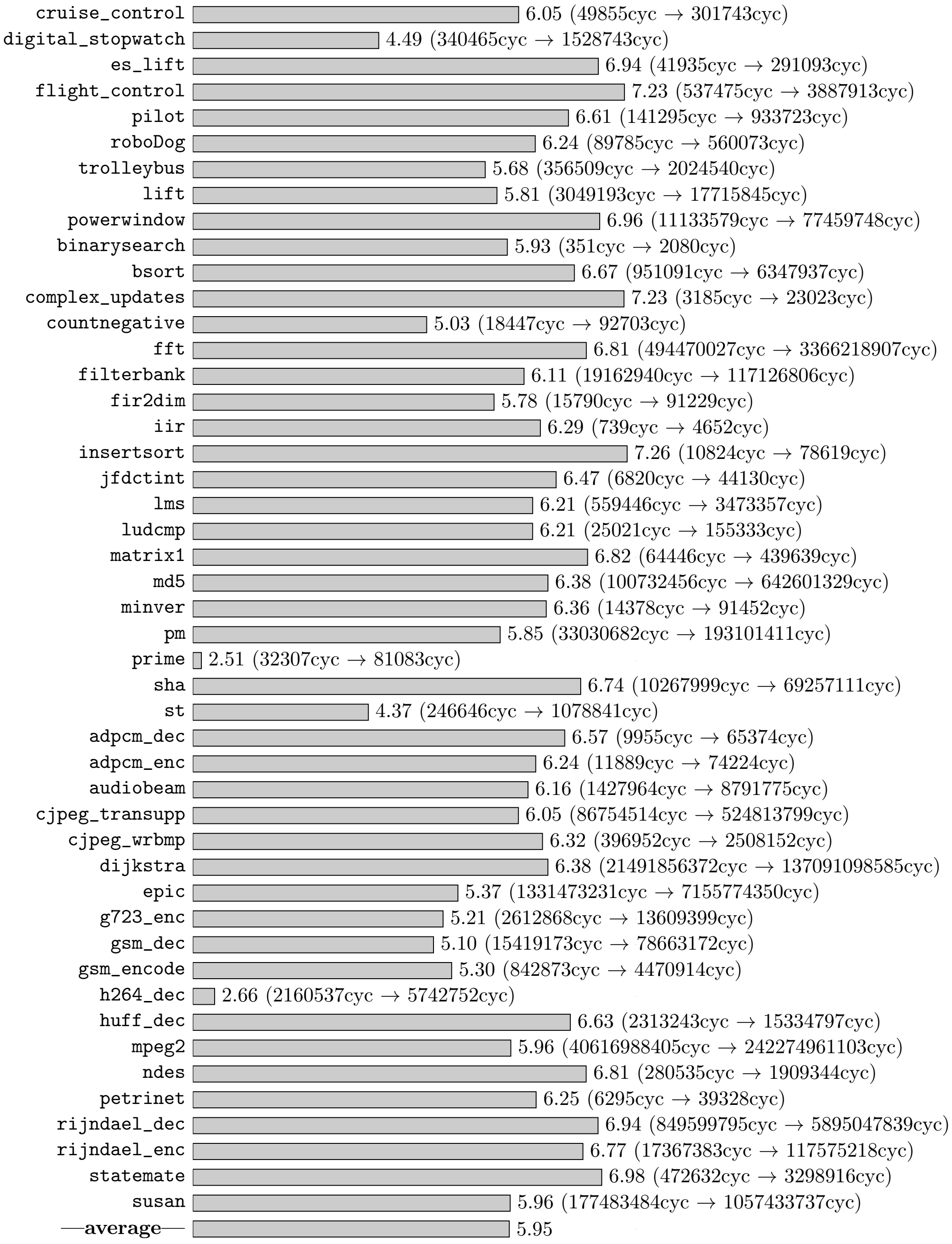


Figure B.18.: Co-runner-insensitive WCET analysis for an octa-core processor with core configuration $Conf_{is}^{io}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

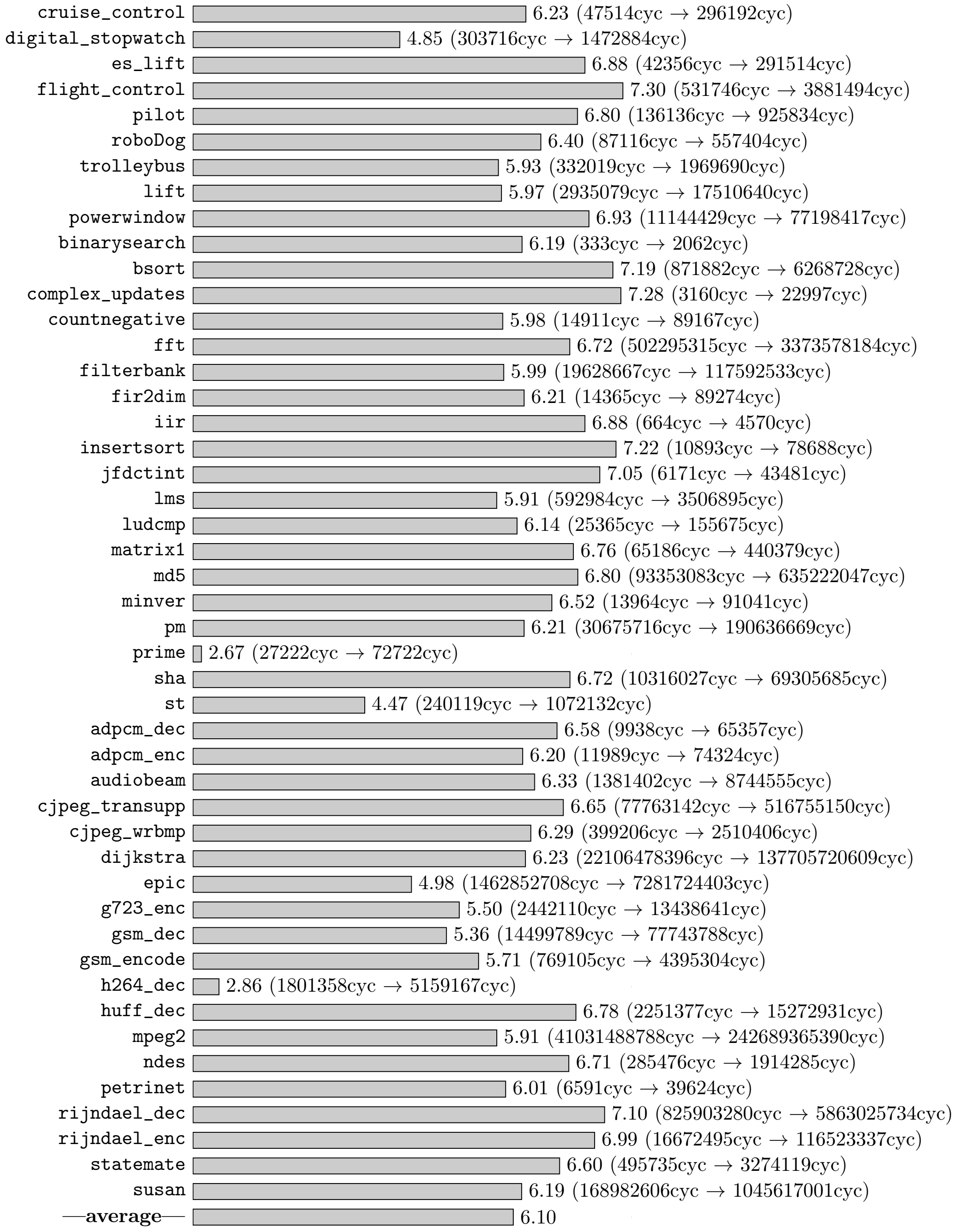


Figure B.19.: Co-runner-insensitive WCET analysis for an octa-core processor with core configuration $Conf_{is}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

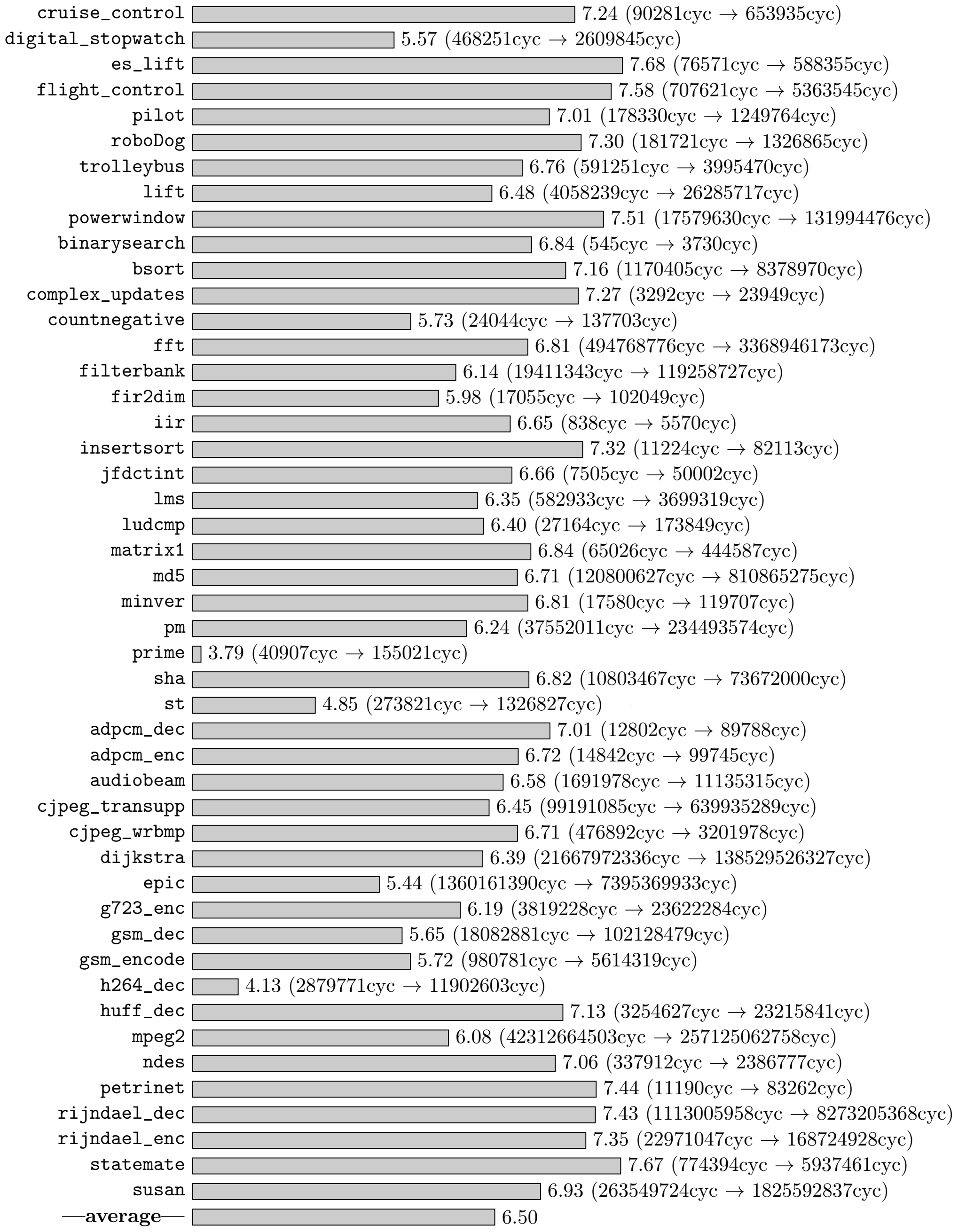


Figure B.20.: Co-runner-insensitive WCET analysis for an octa-core processor with core configuration $Conf_{ic}^{io}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

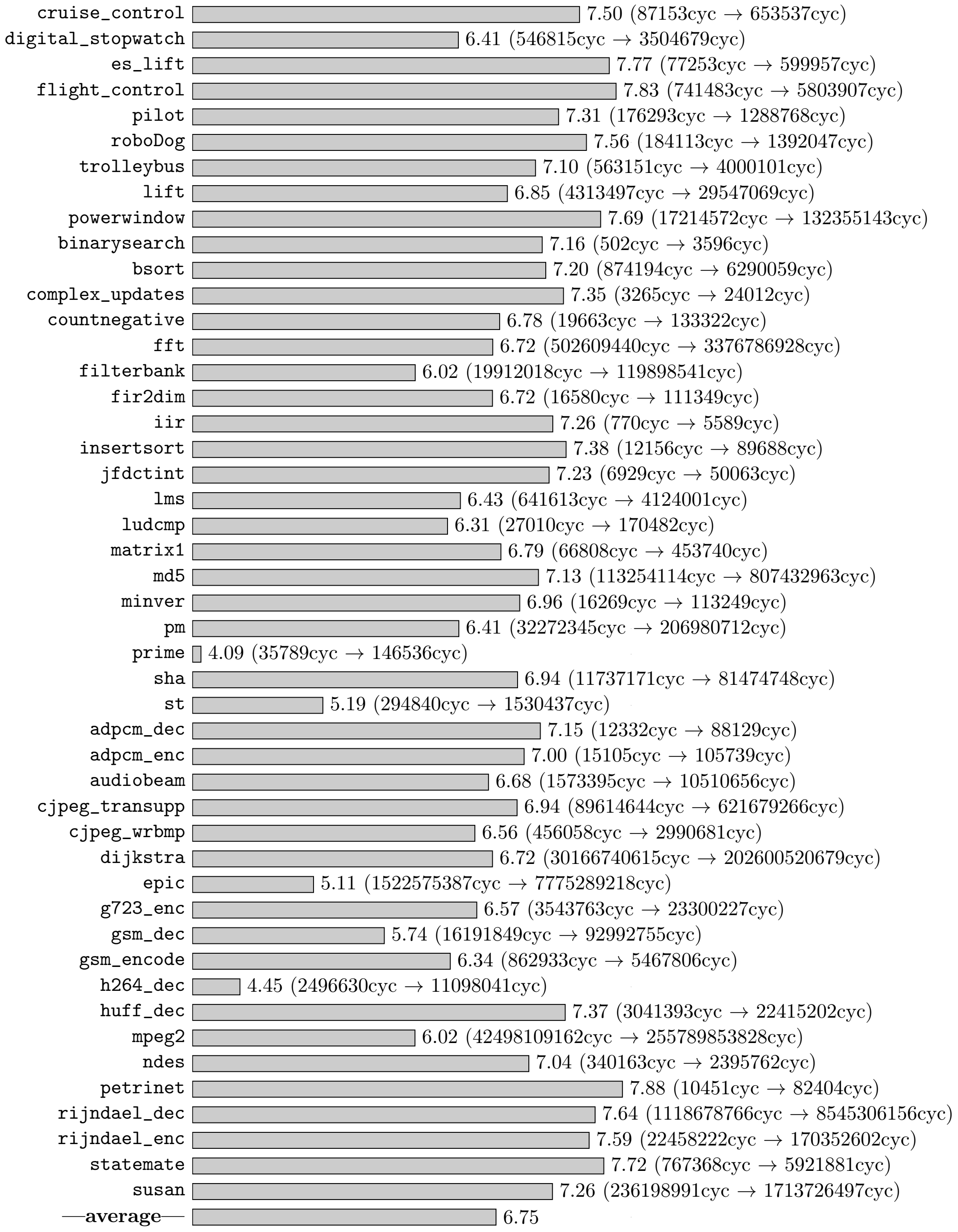


Figure B.21.: Co-runner-insensitive WCET analysis for an octa-core processor with core configuration $Conf_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of an analysis assuming the absence of shared-bus interference.

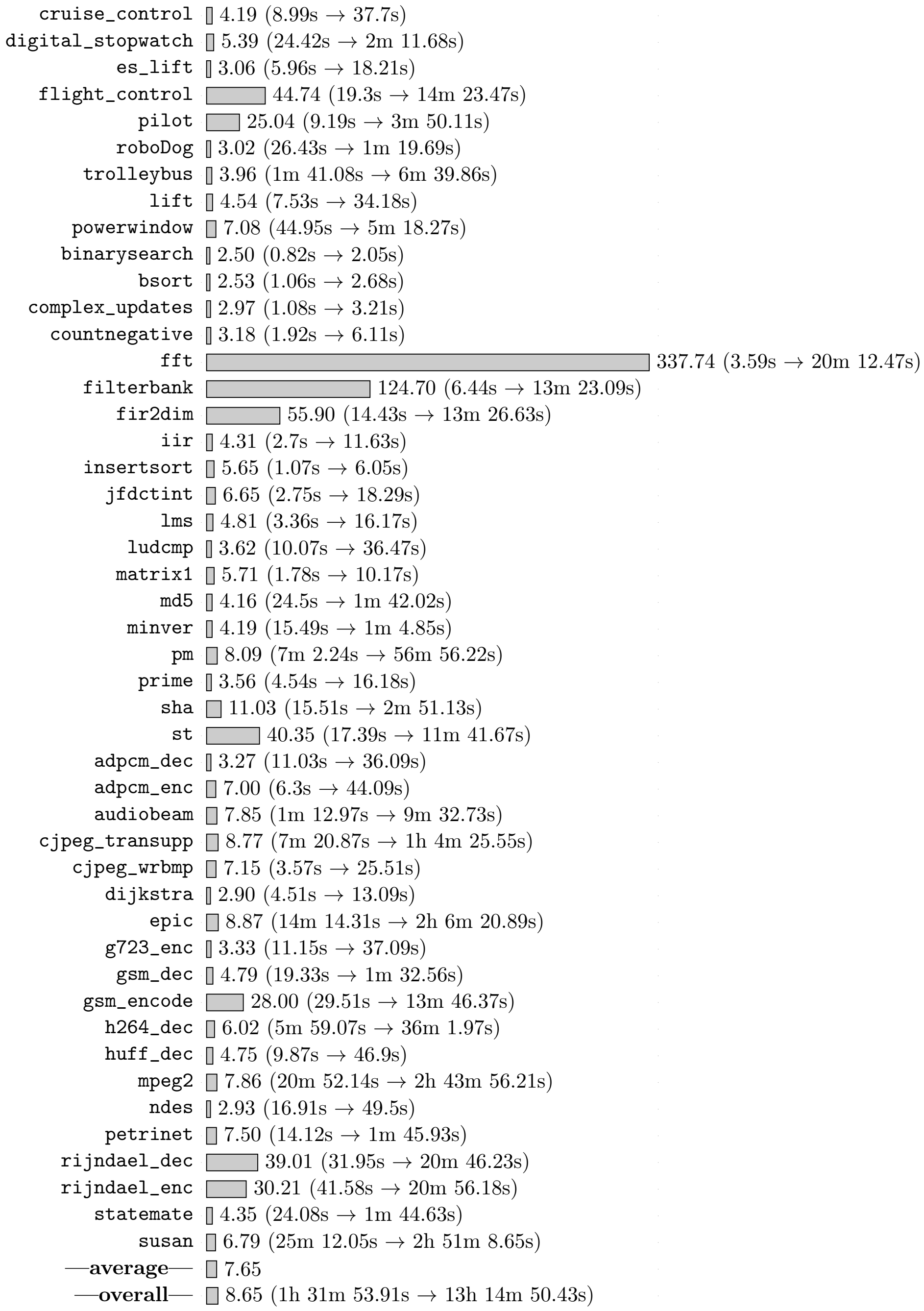


Figure B.22.: Arrival curve values calculated by implicit subpath enumeration ($Bound^{impli}_{1,E}(l)$): *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs.

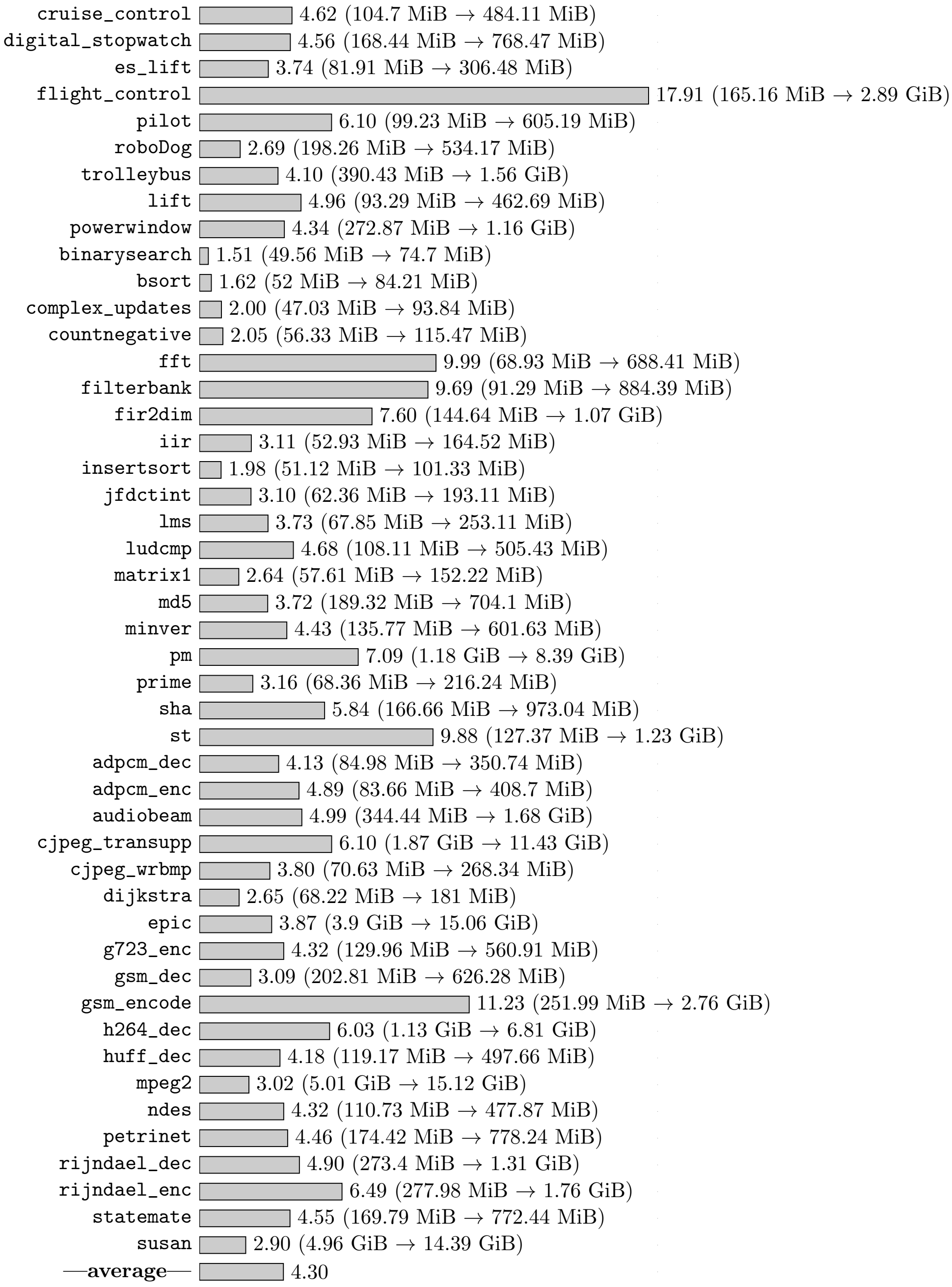


Figure B.23.: Arrival curve values calculated by implicit subpath enumeration ($Bound_{1,E}^{impli}(l)$): *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs.

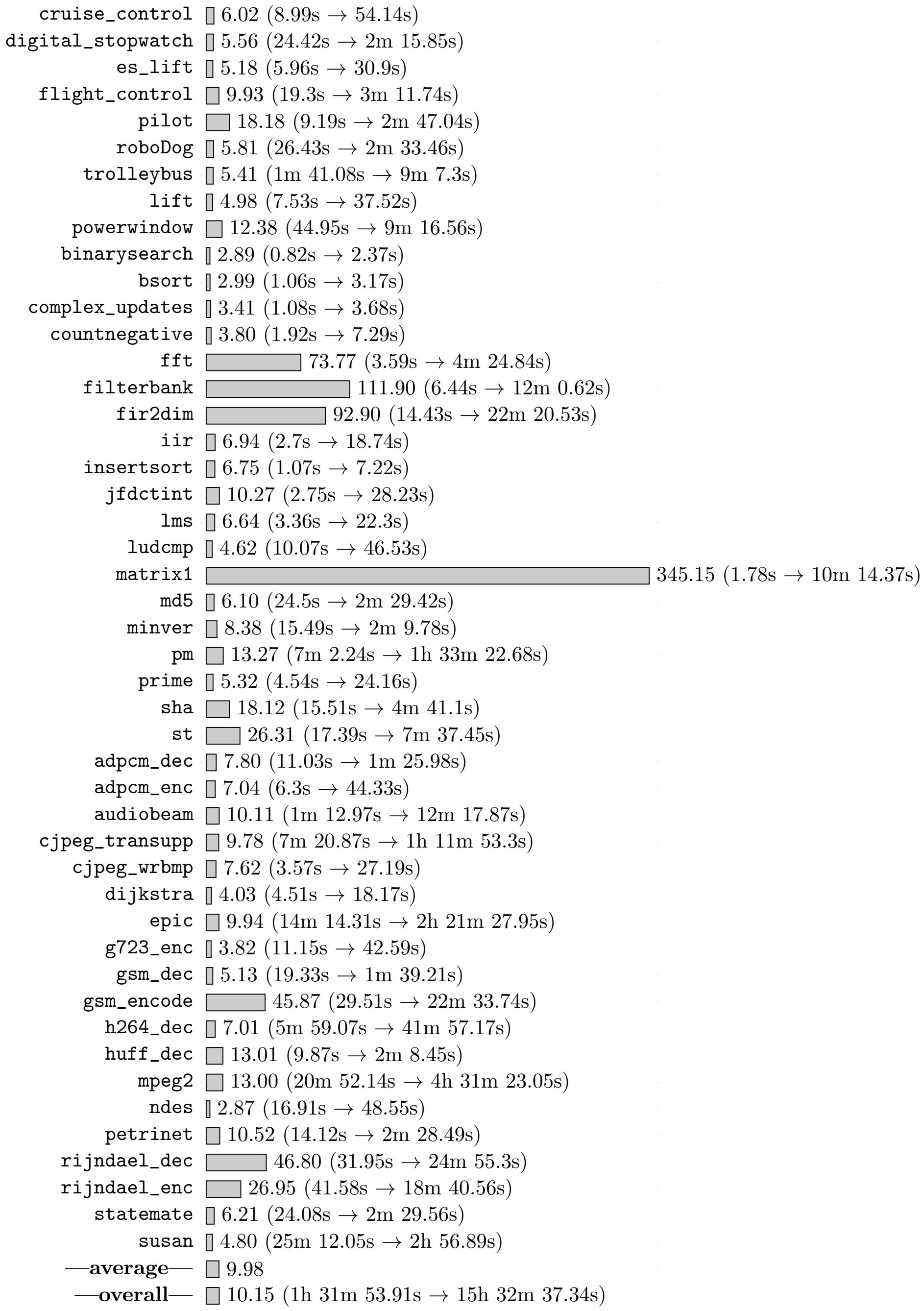


Figure B.24.: Arrival curve values calculated by implicit subpath enumeration ($Bound^{impli}_{comb,E}(l)$): *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs.

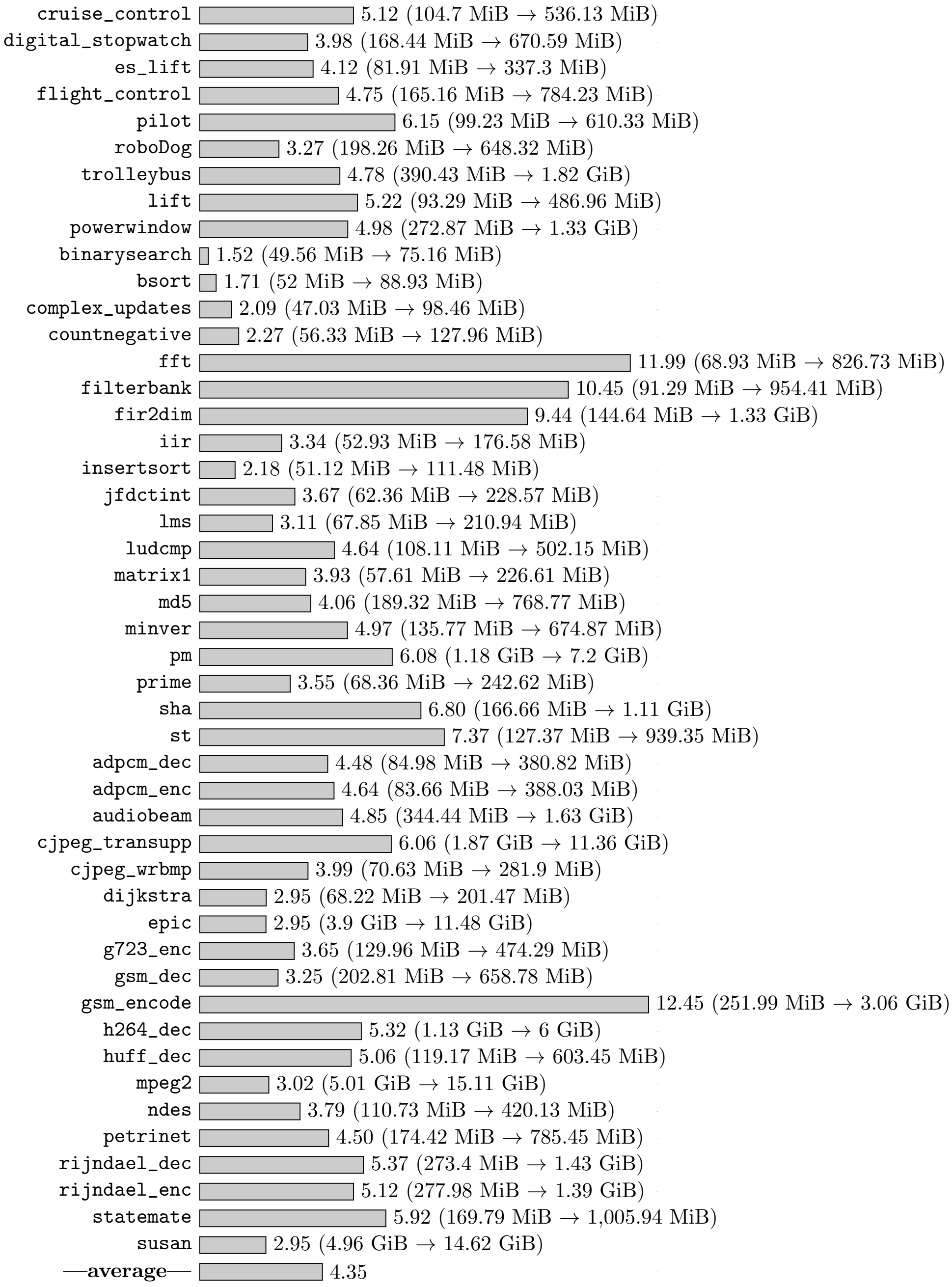


Figure B.25.: Arrival curve values calculated by implicit subpath enumeration ($Bound^{impli}_{comb,E}(l)$): *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs.

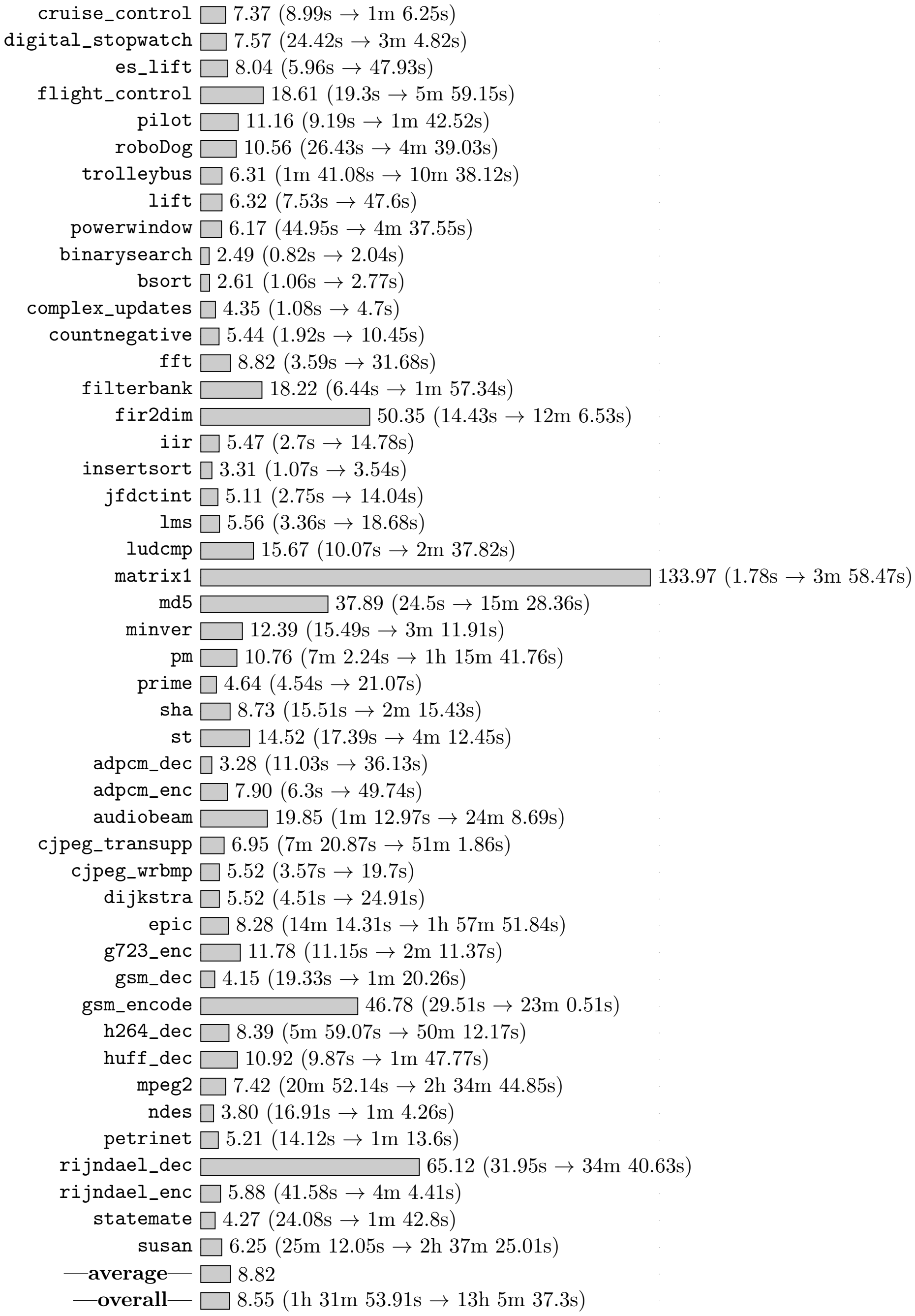


Figure B.26.: Arrival curve values calculated by implicit subpath enumeration without binary variables ($Bound_{1,E}^{impli\prime}(l)$): *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs.

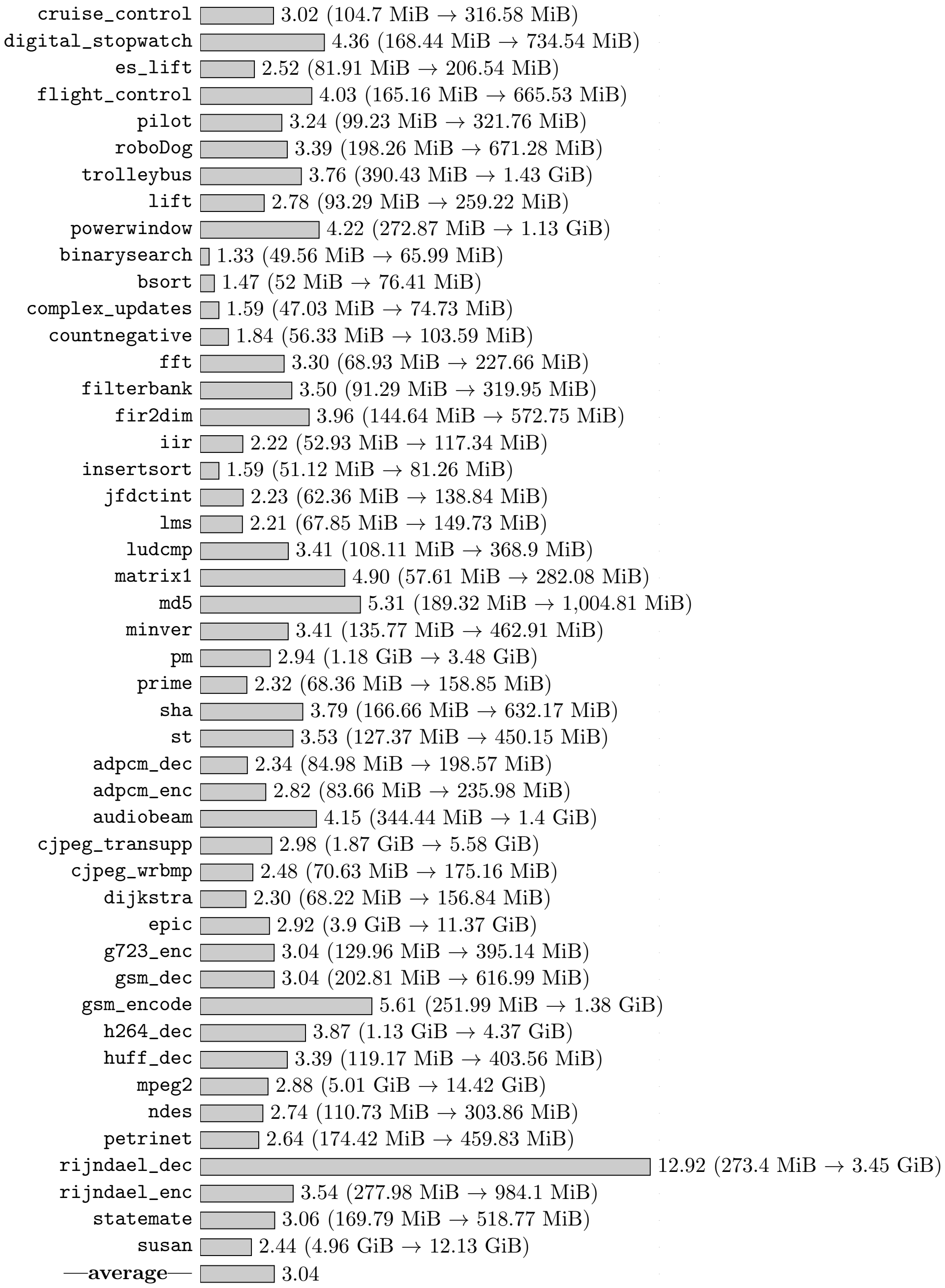


Figure B.27.: Arrival curve values calculated by implicit subpath enumeration without binary variables ($Bound^{impli\prime}_{1,E}(l)$): *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs.

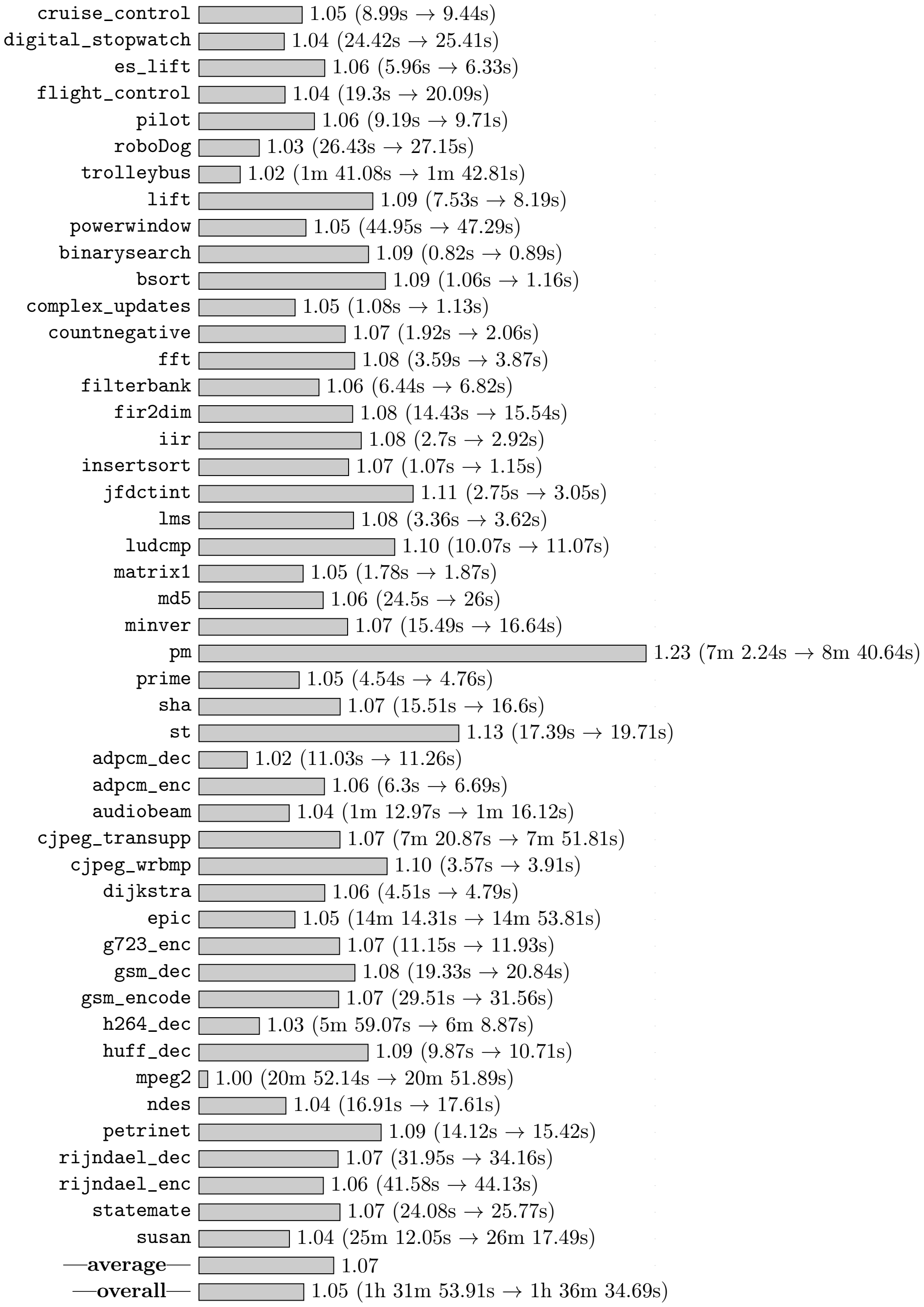


Figure B.28.: Arrival curve values calculated at the granularity of program runs assuming a relative minimum inter-start time of 0.5: *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

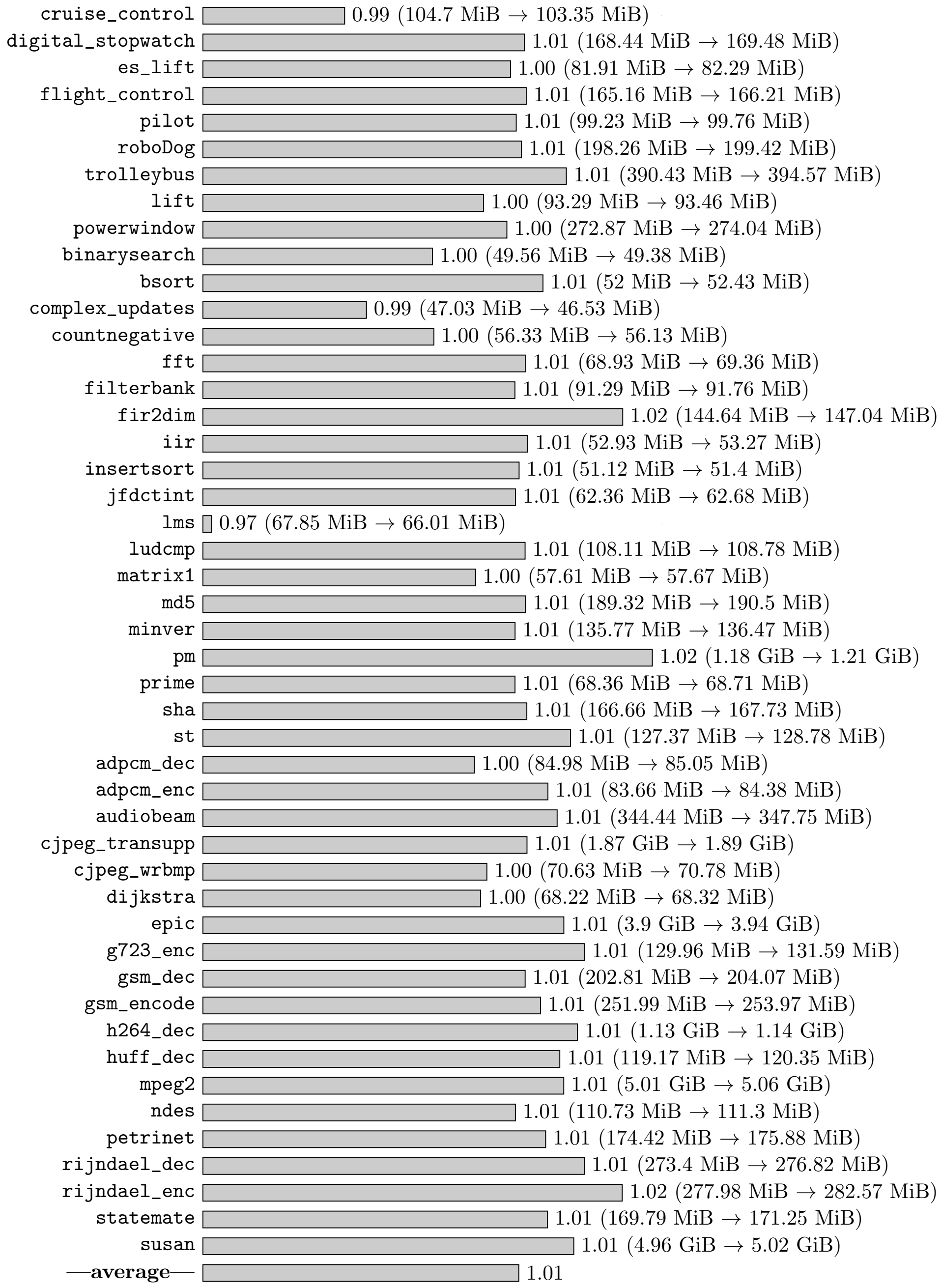


Figure B.29.: Arrival curve values calculated at the granularity of program runs assuming a relative minimum inter-start time of 0.5: *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

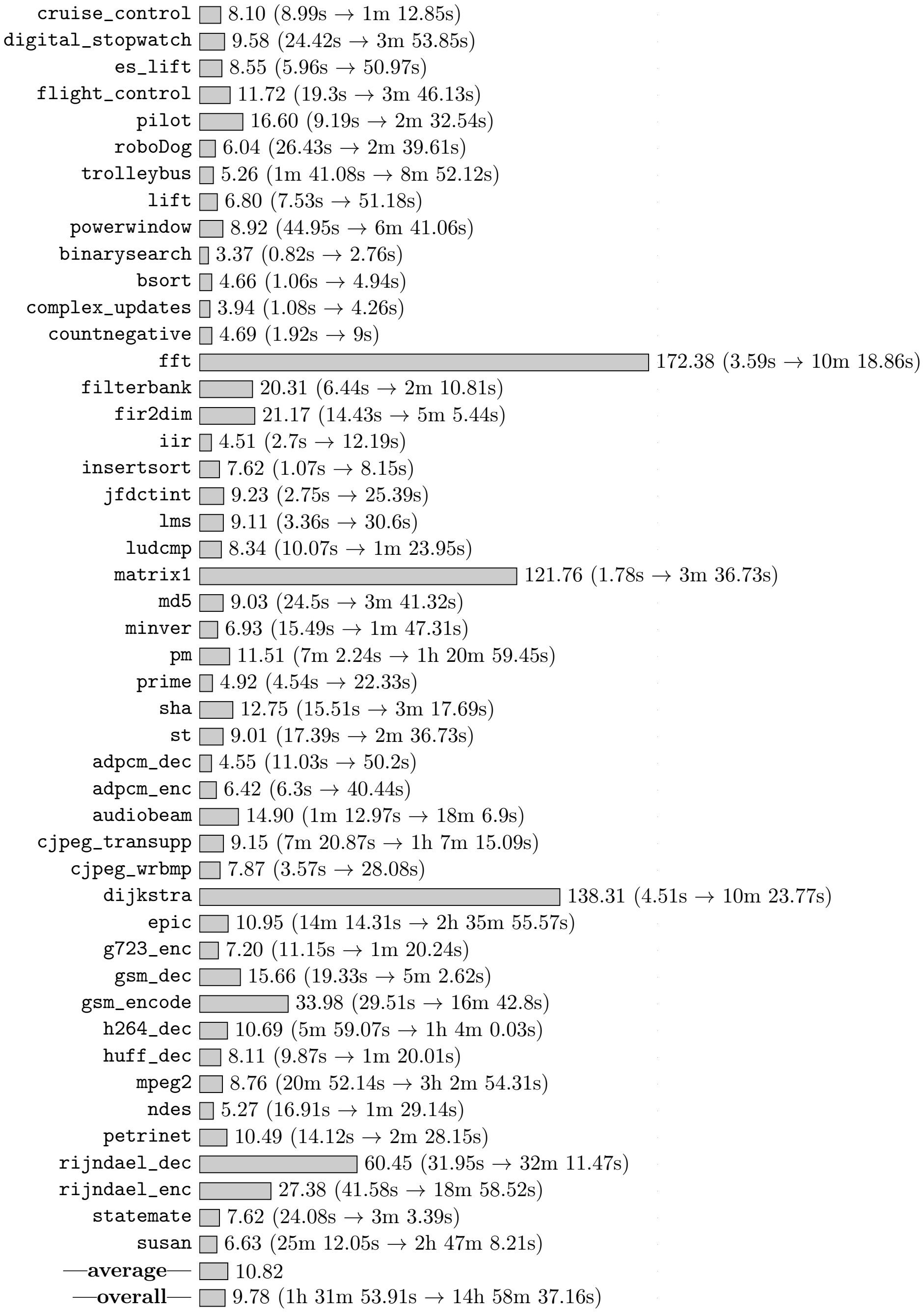


Figure B.30.: Arrival curve values calculated by implicit subpath enumeration assuming a relative minimum inter-start time of 0.5: *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

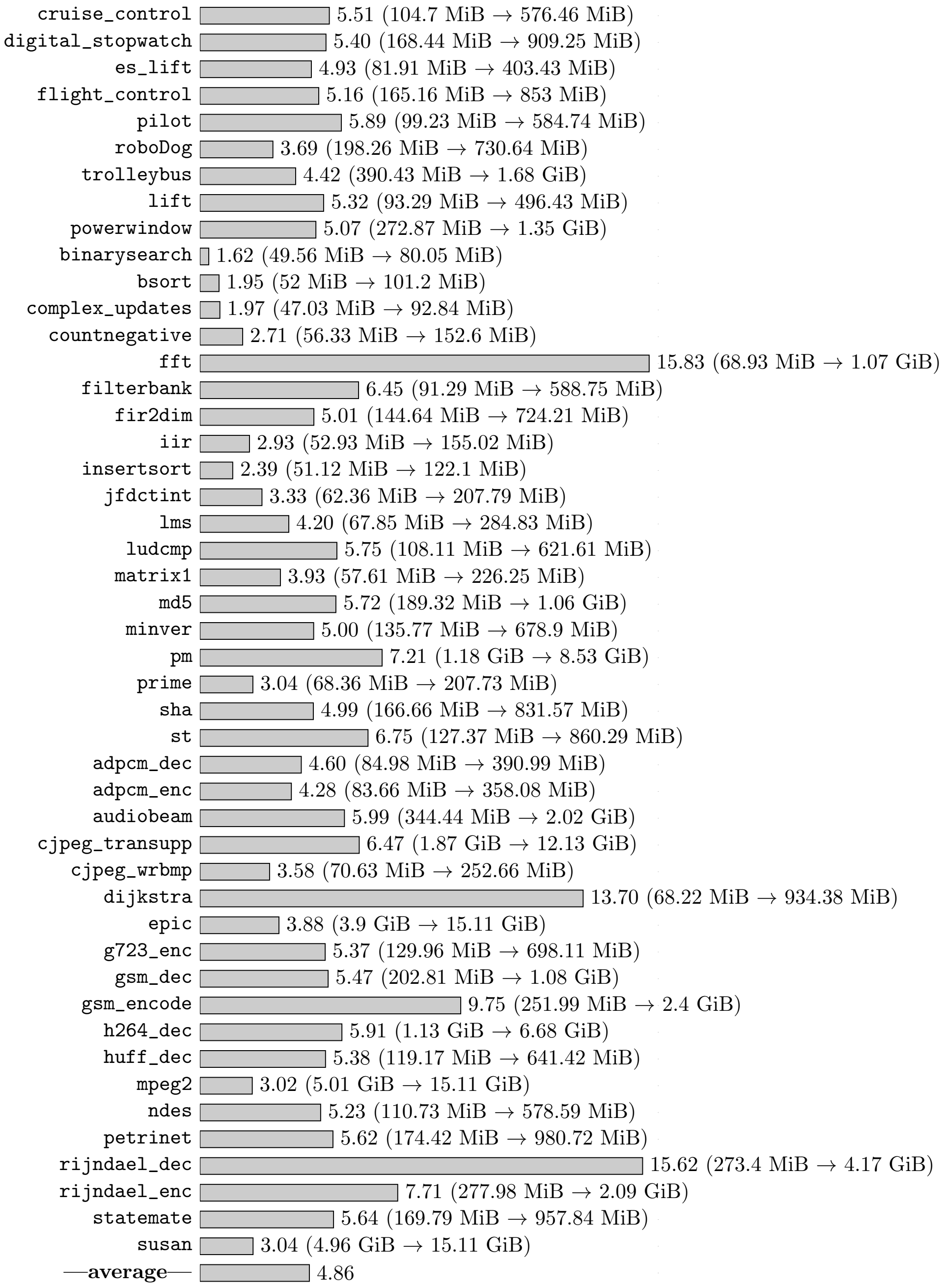


Figure B.31.: Arrival curve values calculated by implicit subpath enumeration assuming a relative minimum inter-start time of 0.5: *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

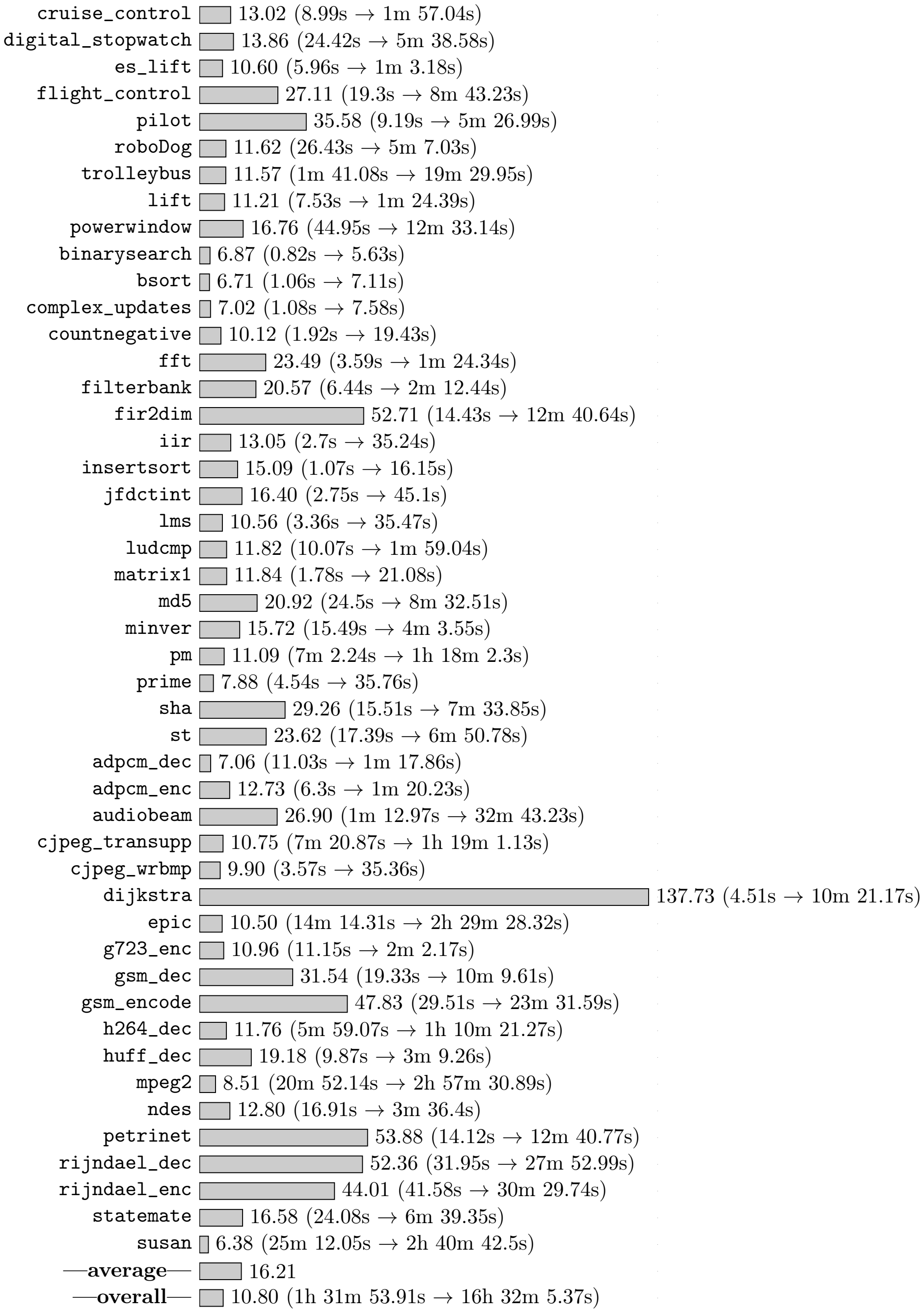


Figure B.32.: Arrival curve values calculated by implicit subpath enumeration incorporating a relative minimum inter-start time of 0.5 in a more precise way: *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

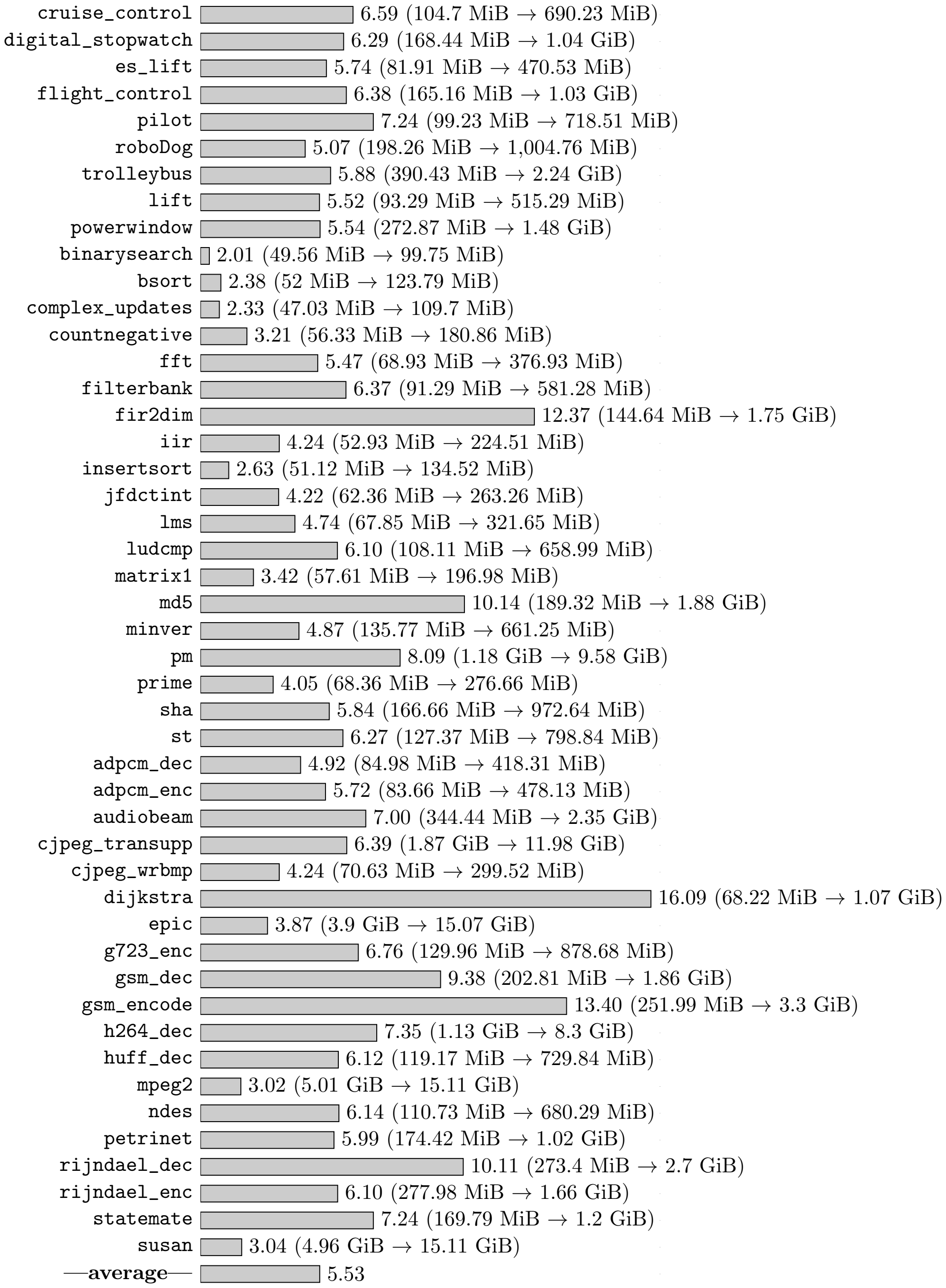


Figure B.33.: Arrival curve values calculated by implicit subpath enumeration incorporating a relative minimum inter-start time of 0.5 in a more precise way: *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

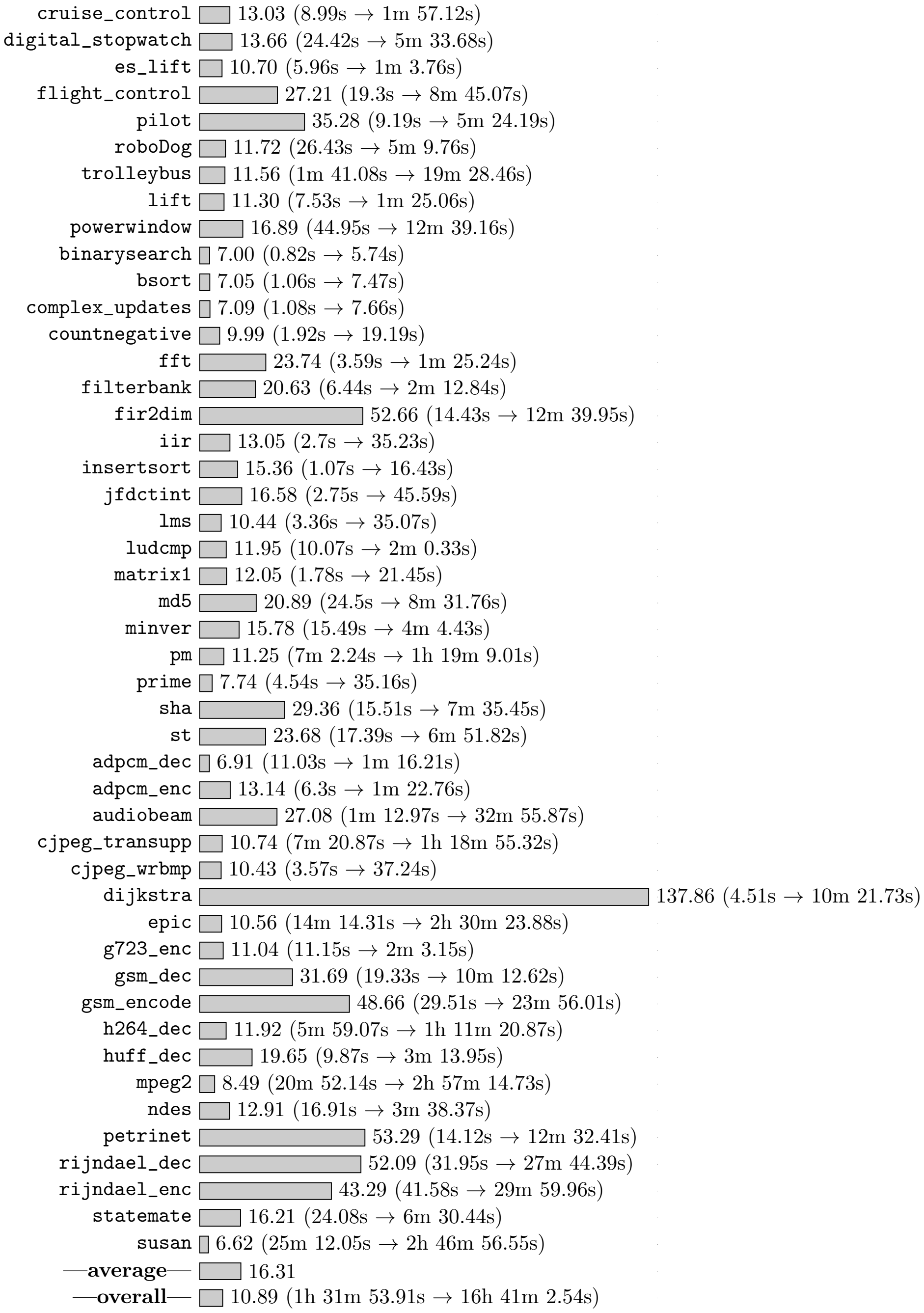


Figure B.34.: Arrival curve values calculated by taking the minimum of the curve values calculated by the approaches presented in Section 10.3.1 and Section 10.3.3: *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

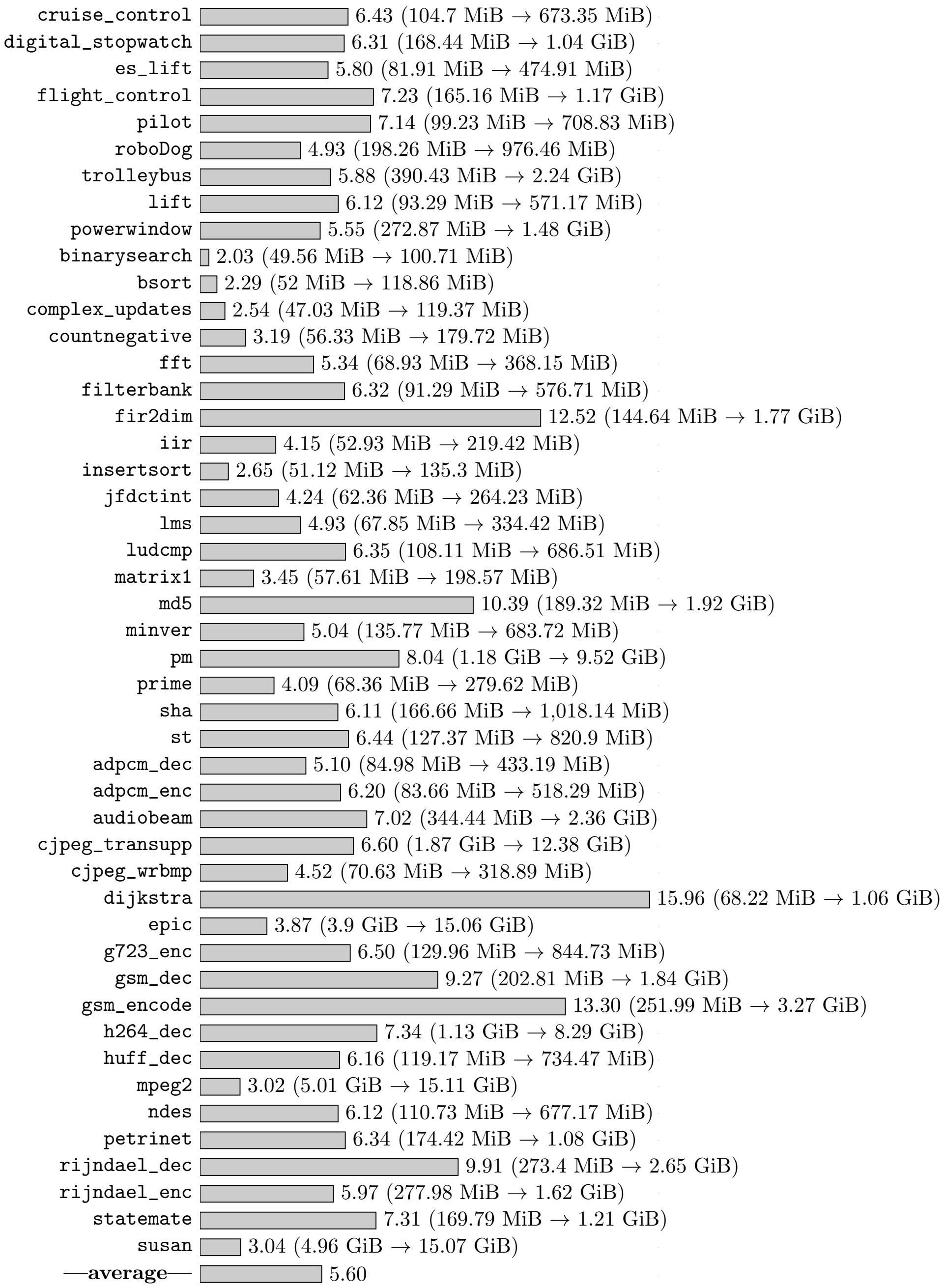


Figure B.35.: Arrival curve values calculated by taking the minimum of the curve values calculated by the approaches presented in Section 10.3.1 and Section 10.3.3: *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

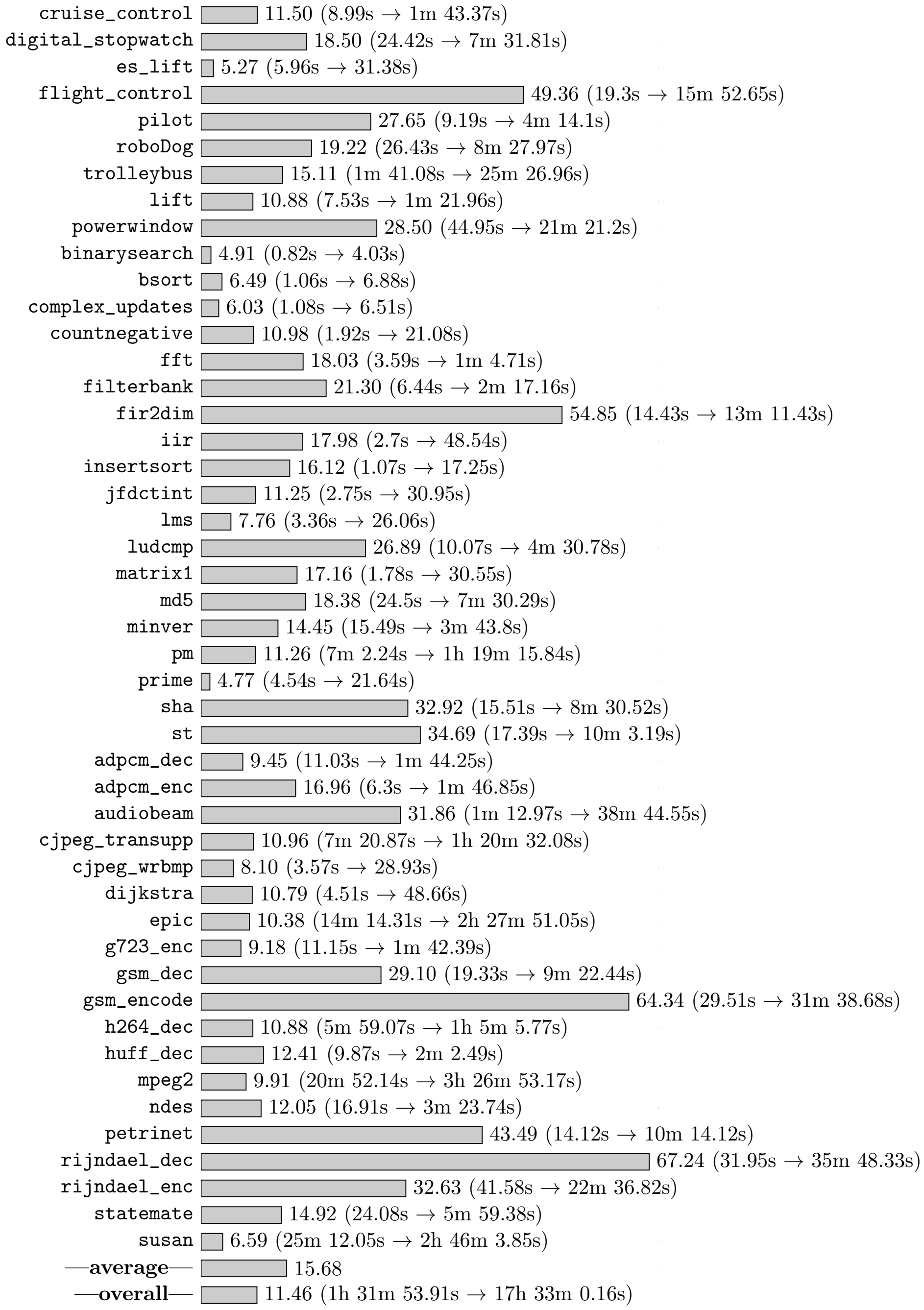


Figure B.36.: Arrival curve values calculated by incorporating the curve value calculated by the approach presented in Section 10.3.1 as upper bound during the calculation presented in Section 10.3.3: *calculation runtime* per benchmark *normalized* to the corresponding runtime of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

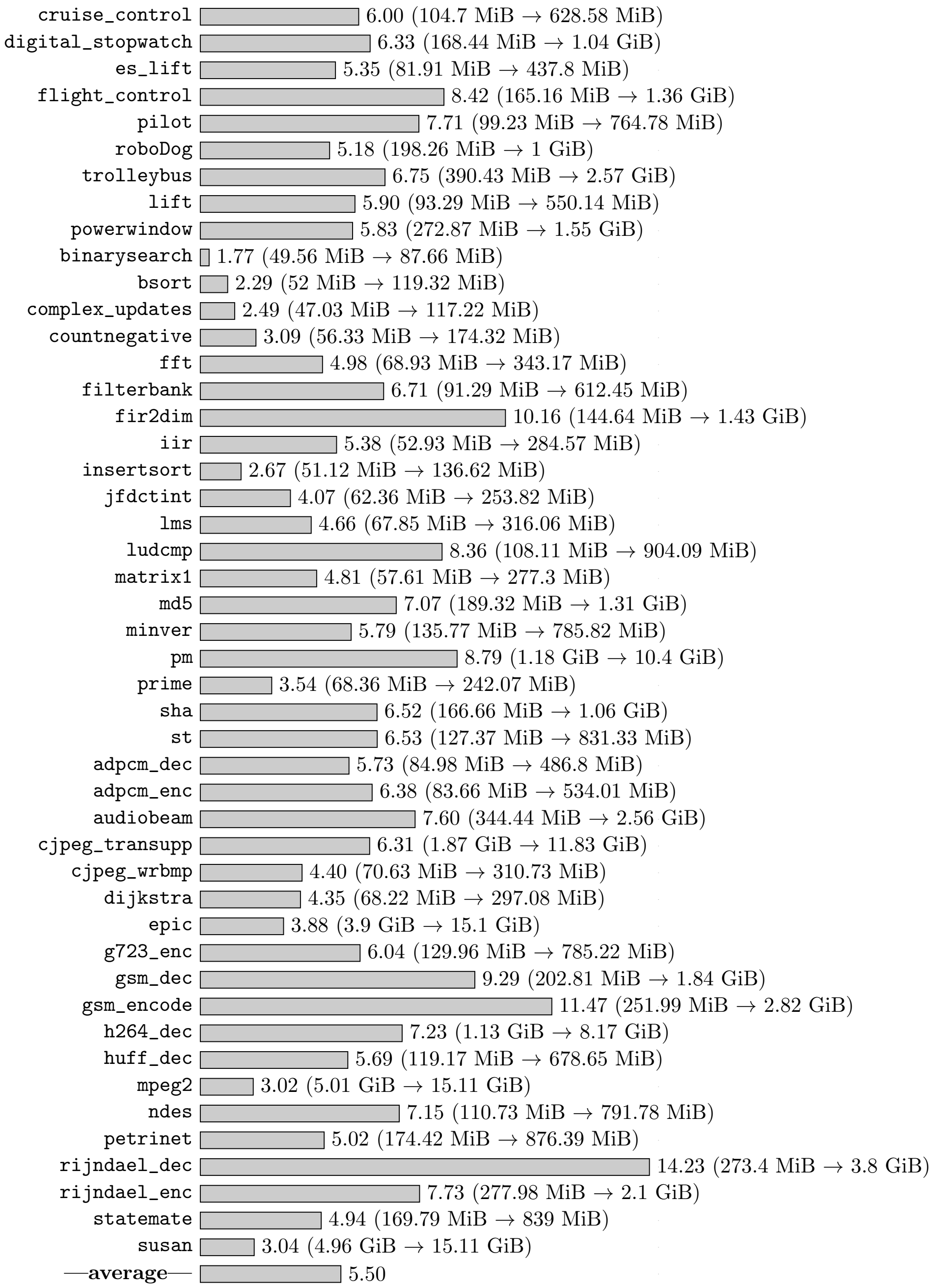


Figure B.37.: Arrival curve values calculated by incorporating the curve value calculated by the approach presented in Section 10.3.1 as upper bound during the calculation presented in Section 10.3.3: *memory consumption* per benchmark *normalized* to the corresponding memory consumption of a calculation at the granularity of program runs not assuming a relative minimum inter-start time.

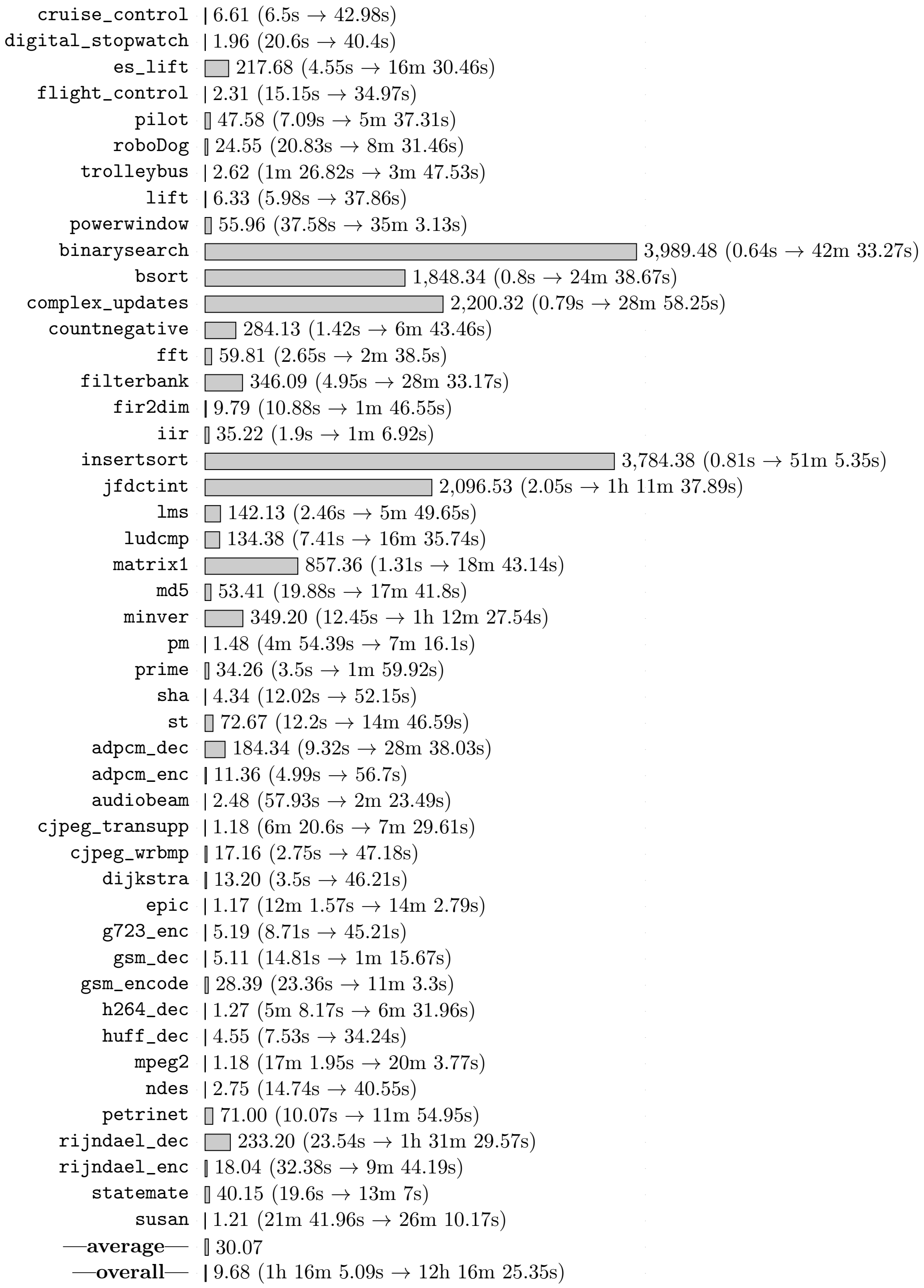


Figure B.38.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *ISPET*, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

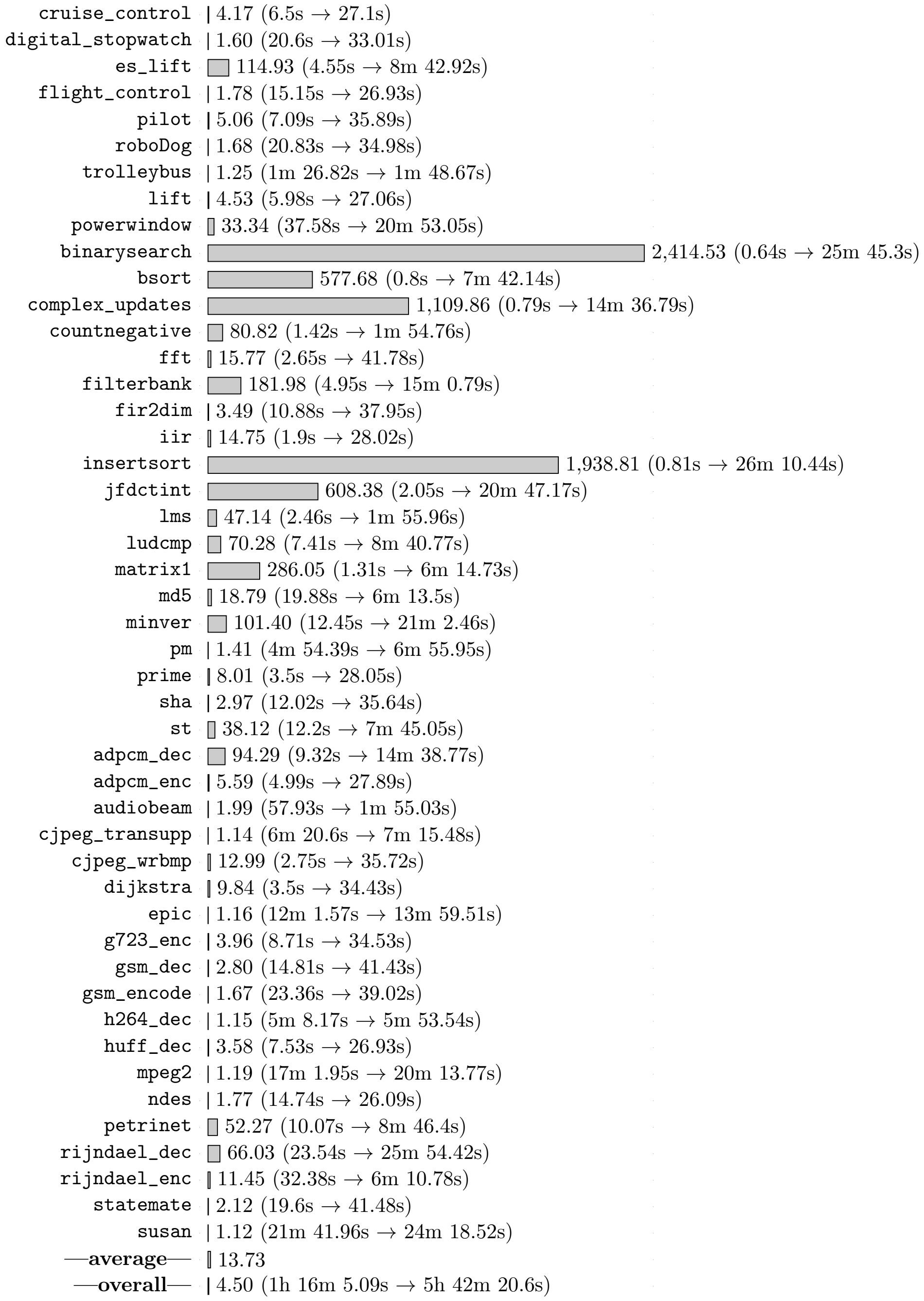


Figure B.39.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $\mathit{progGran}_{0.5}$, cf. Table 10.11) for a quad-core processor with core configuration Conf_{ic}^{ooo}: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

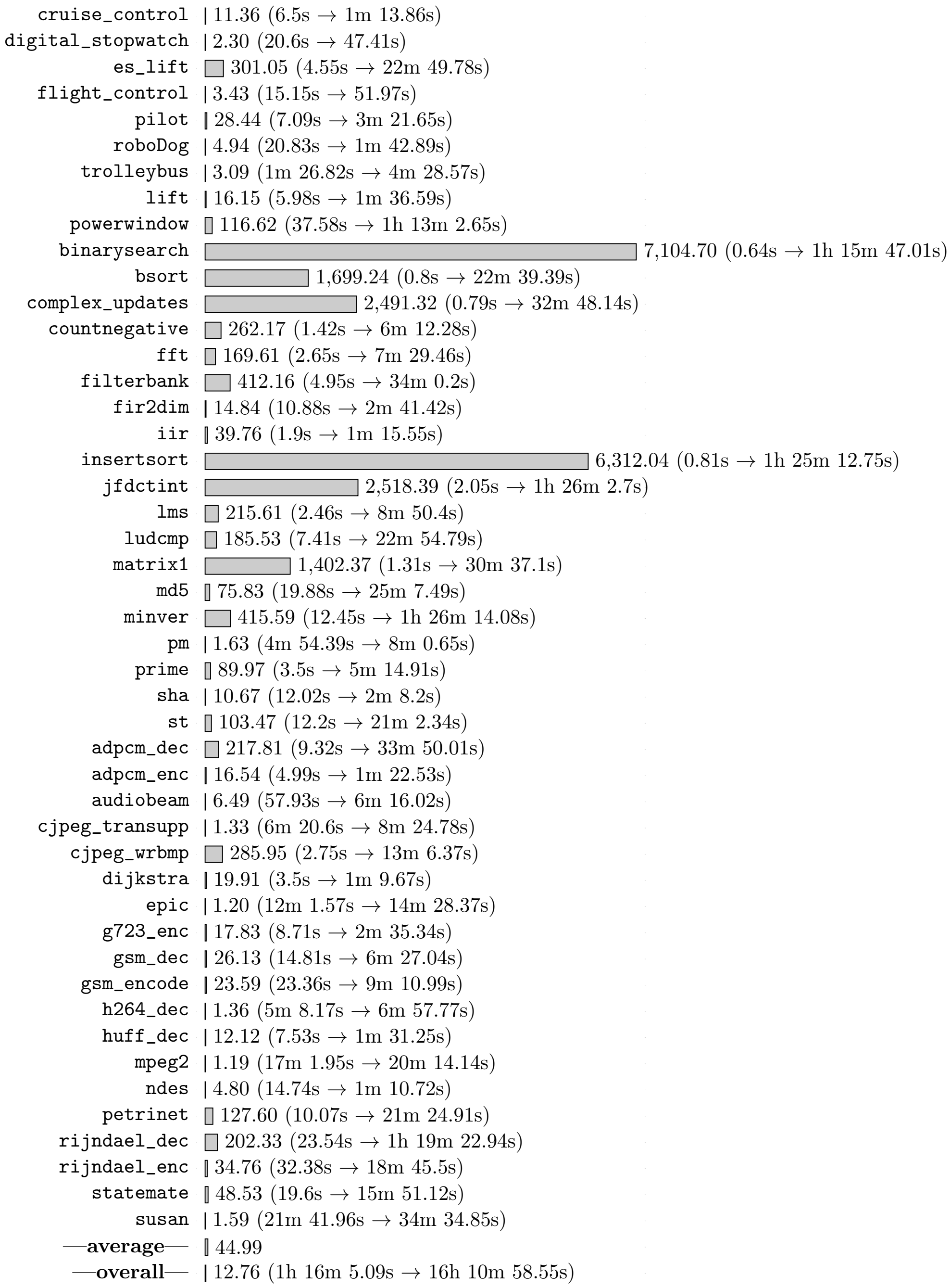


Figure B.40.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $combined_{0.5}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

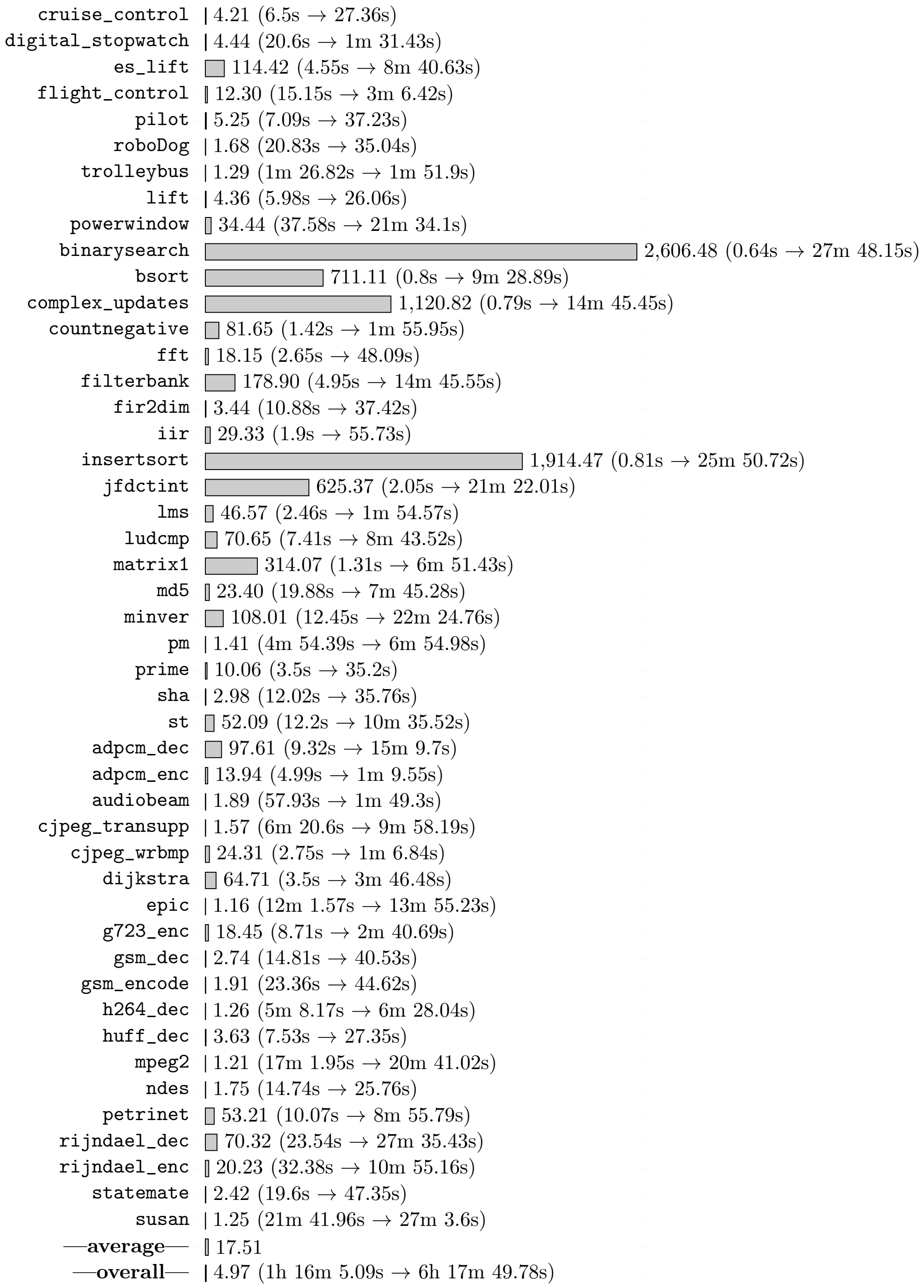


Figure B.41.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *progGran*$_{0.9}$, cf. Table 10.11) for a quad-core processor with core configuration Conf_{ic}^{ooo}: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

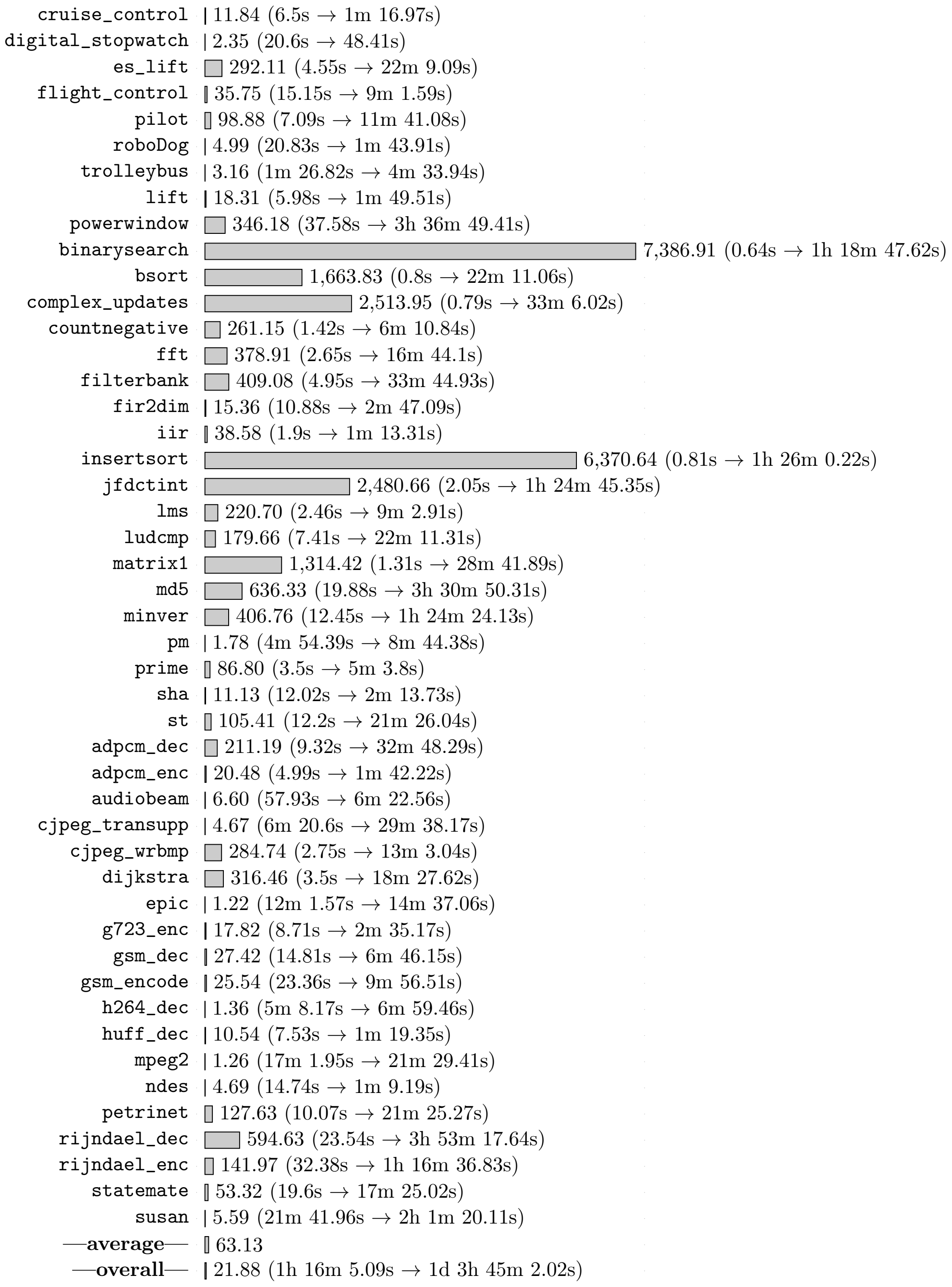


Figure B.42.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $combined_{0.9}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

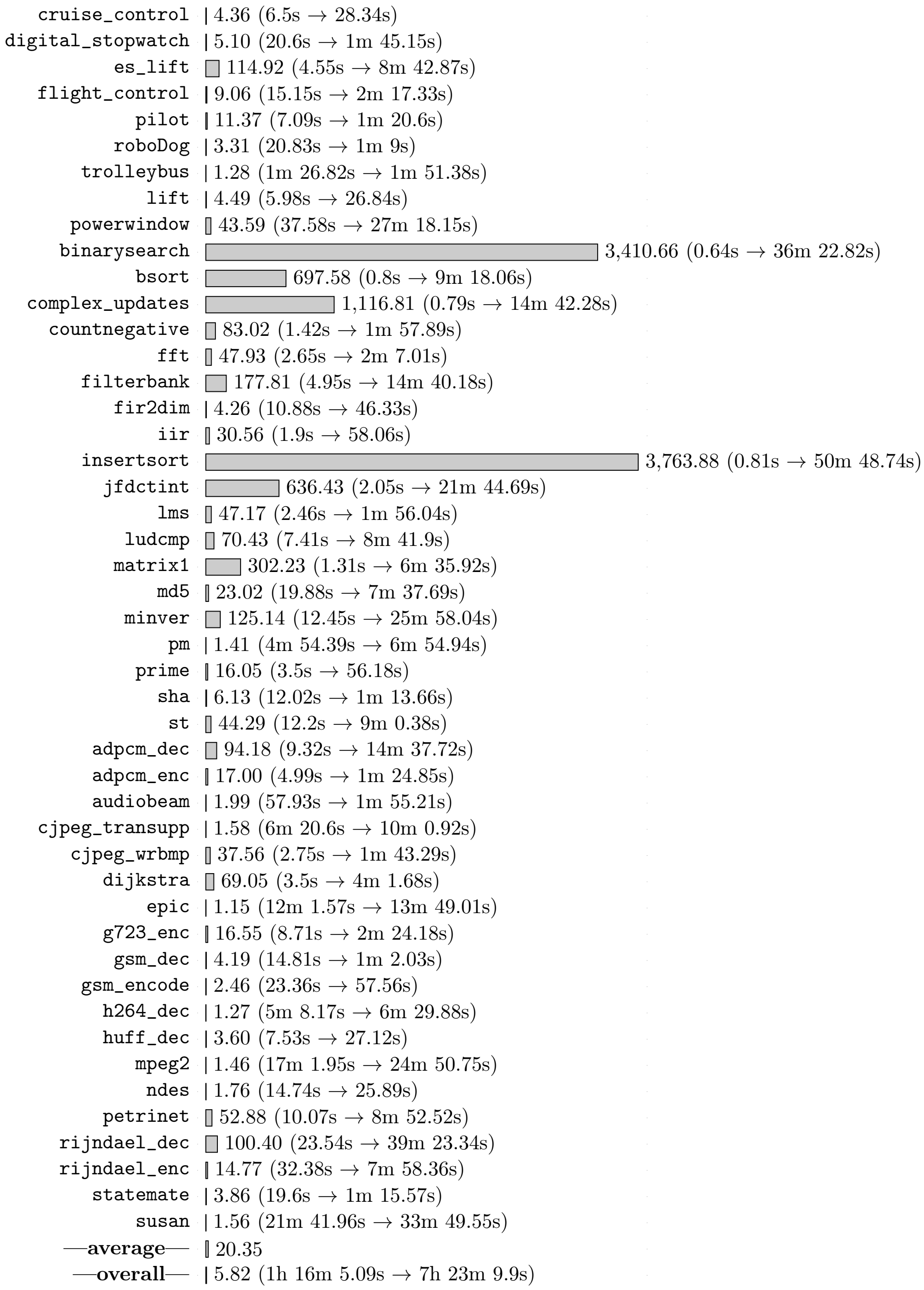


Figure B.43.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *progGran*$_{0.95}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

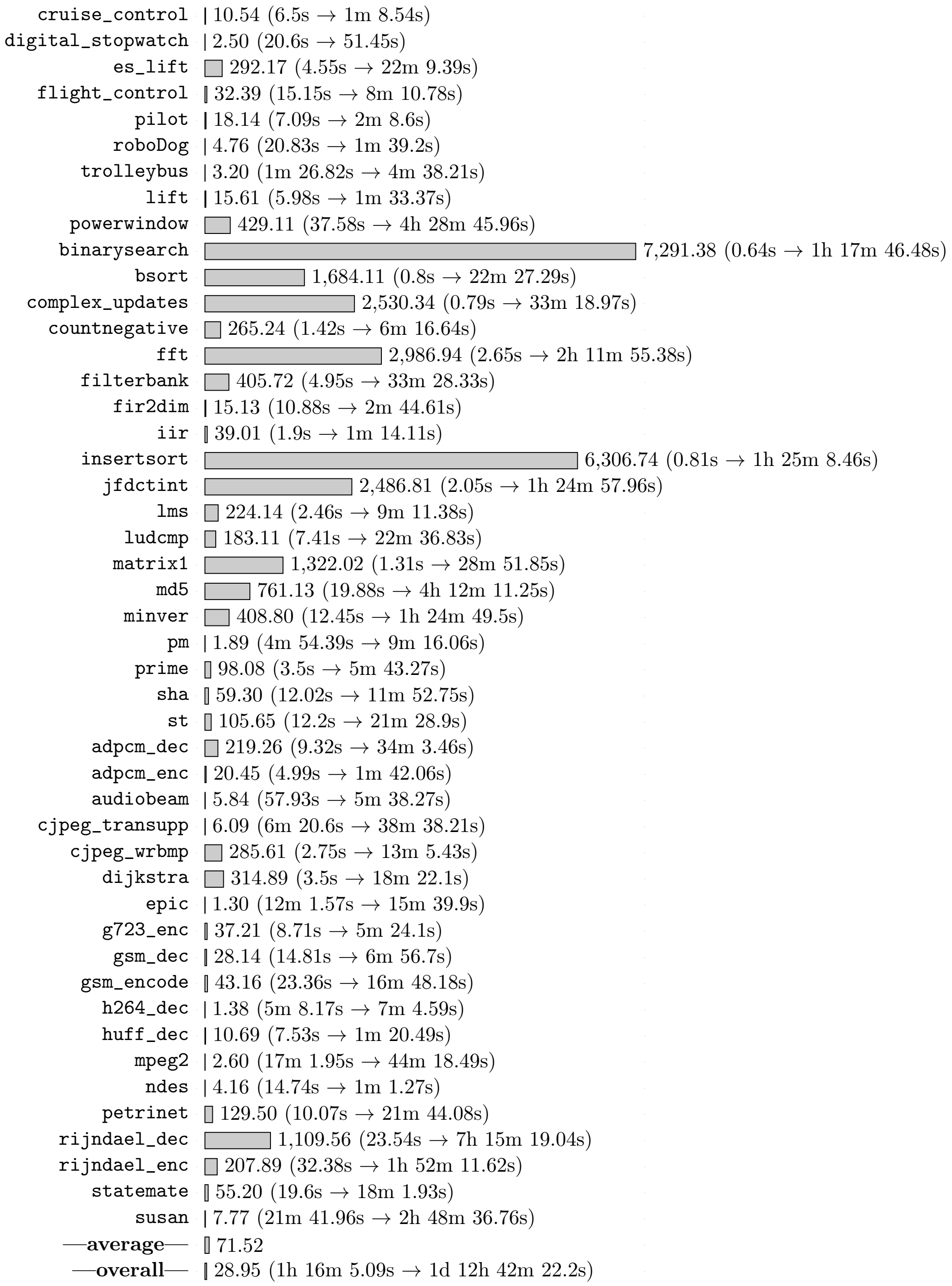


Figure B.44.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $combined_{0.95}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis runtime* per benchmark *normalized* to the corresponding runtime of a co-runner-insensitive analysis.

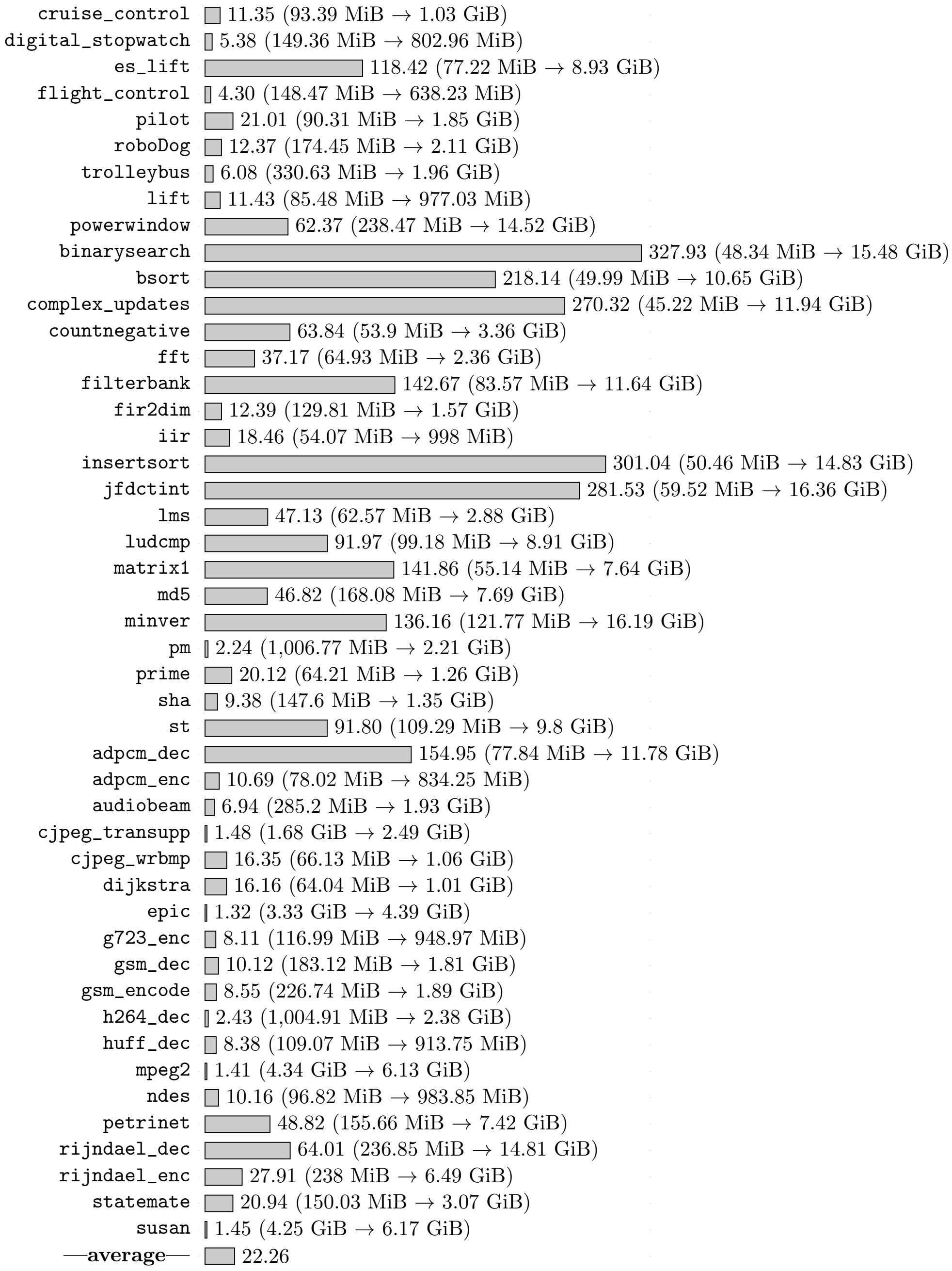


Figure B.45.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *ISPET*, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

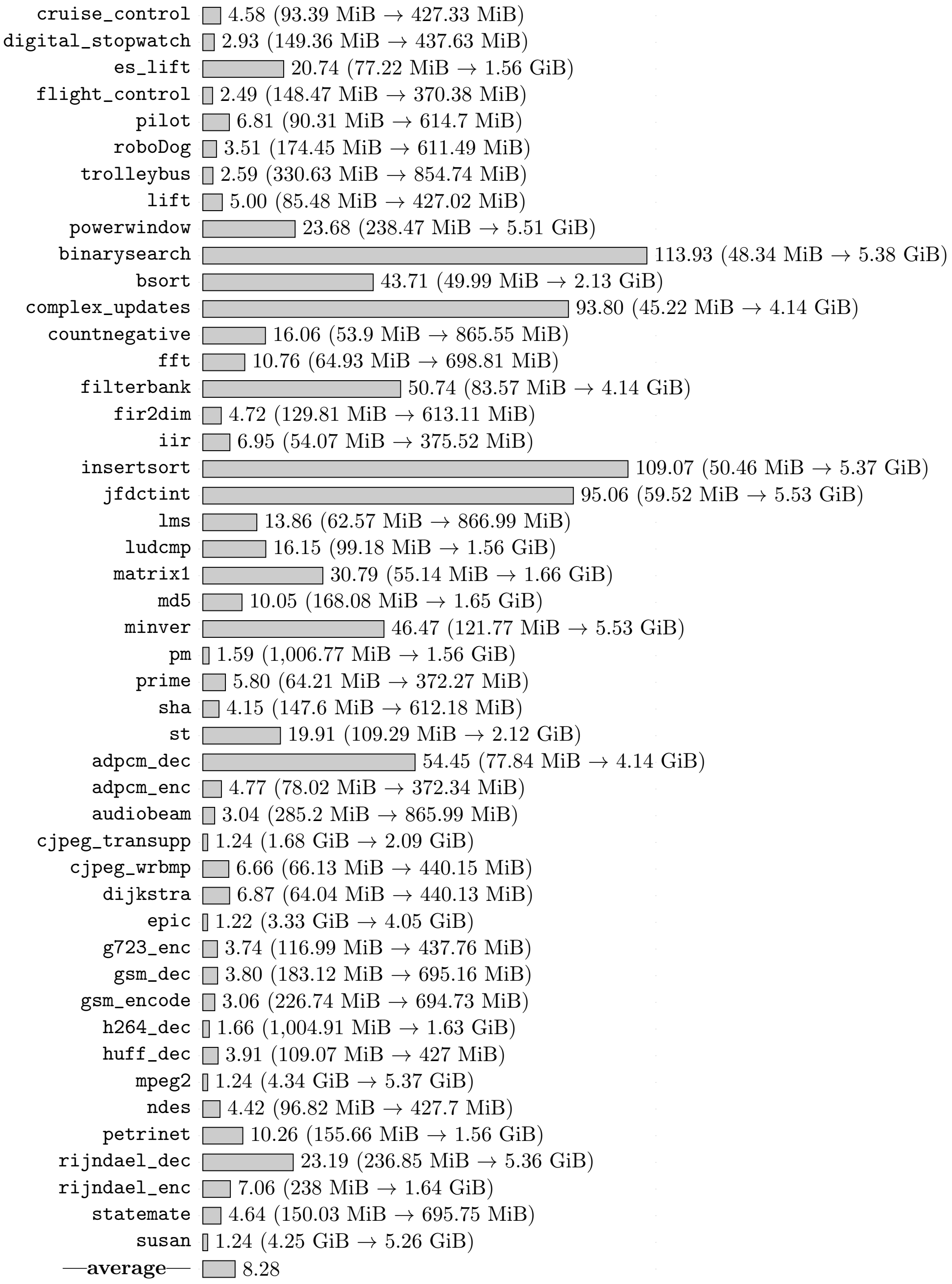


Figure B.46.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *progGran*$_{0.5}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

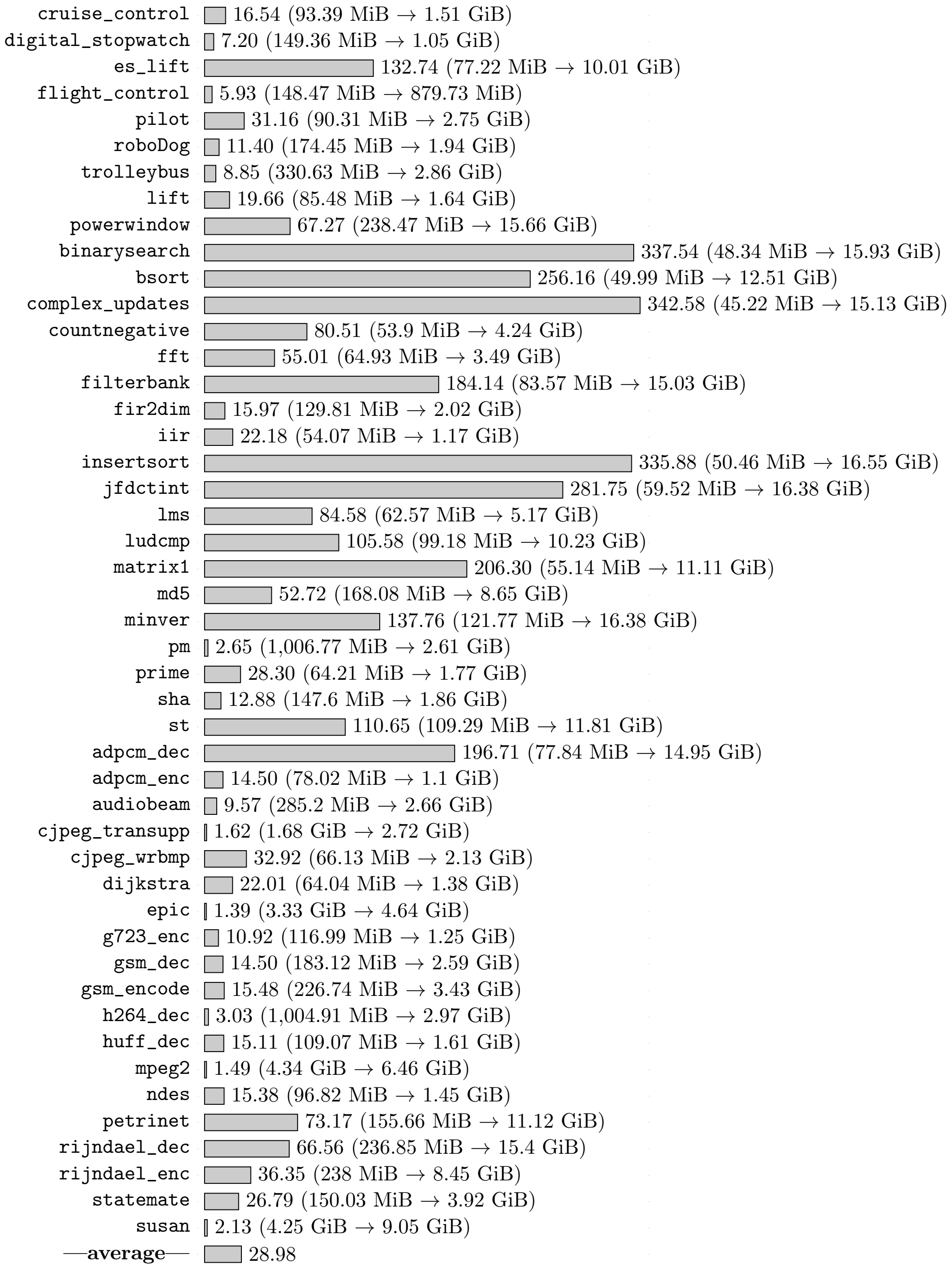


Figure B.47.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *combined*$_{0.5}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

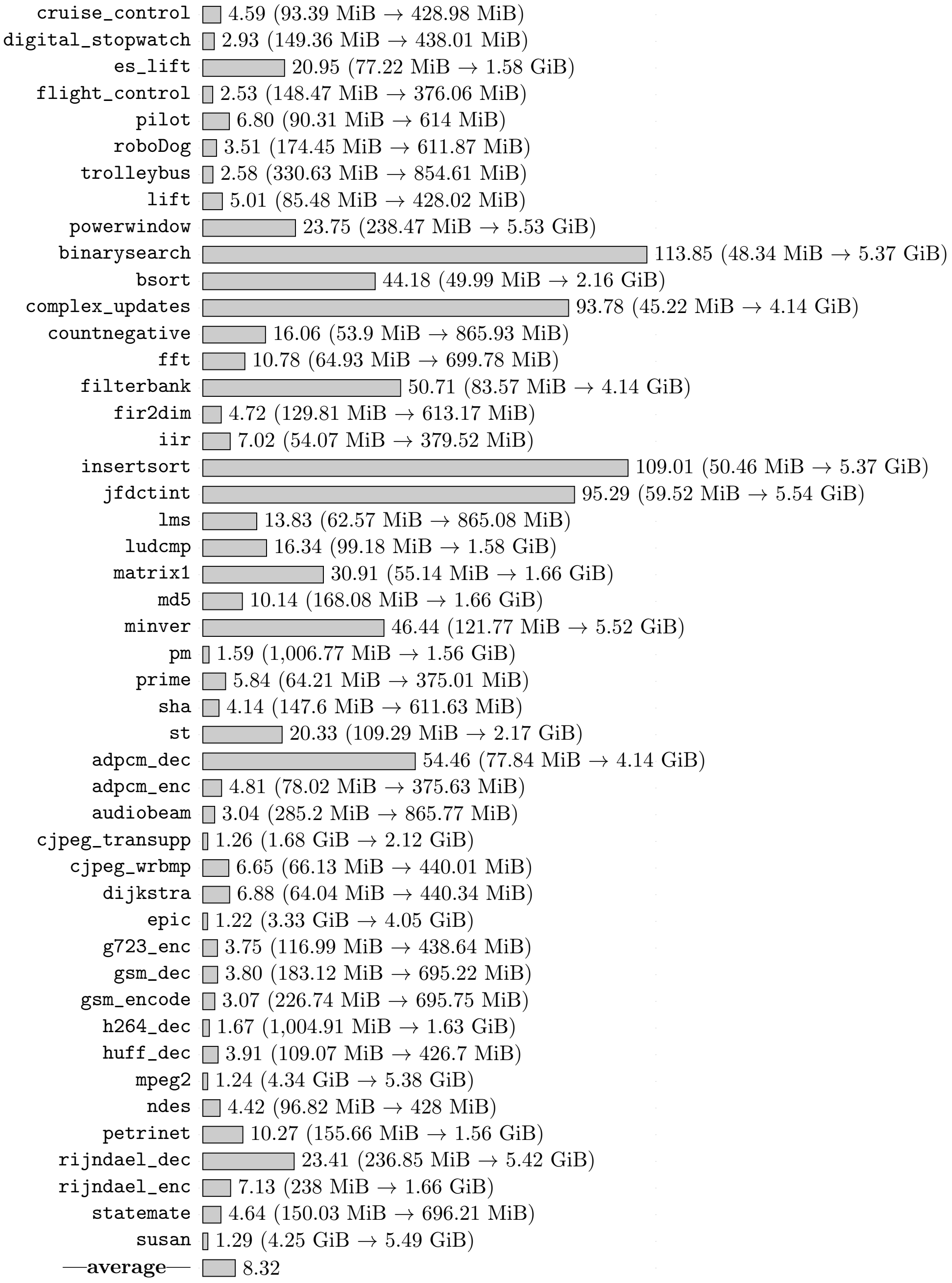


Figure B.48.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *progGran*$_{0.9}$, cf. Table 10.11) for a quad-core processor with core configuration *Conf*$_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

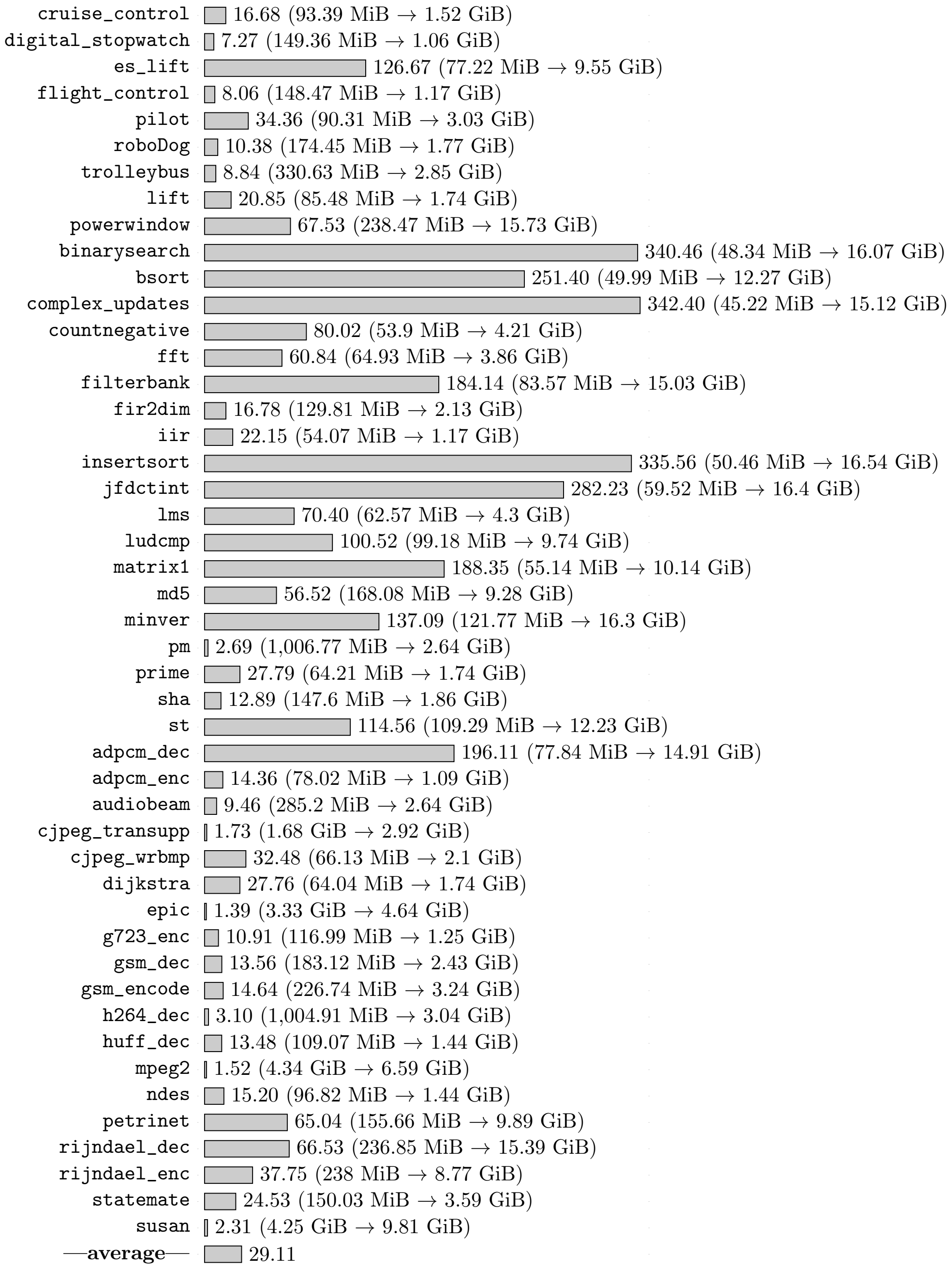


Figure B.49.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *combined*$_{0.9}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

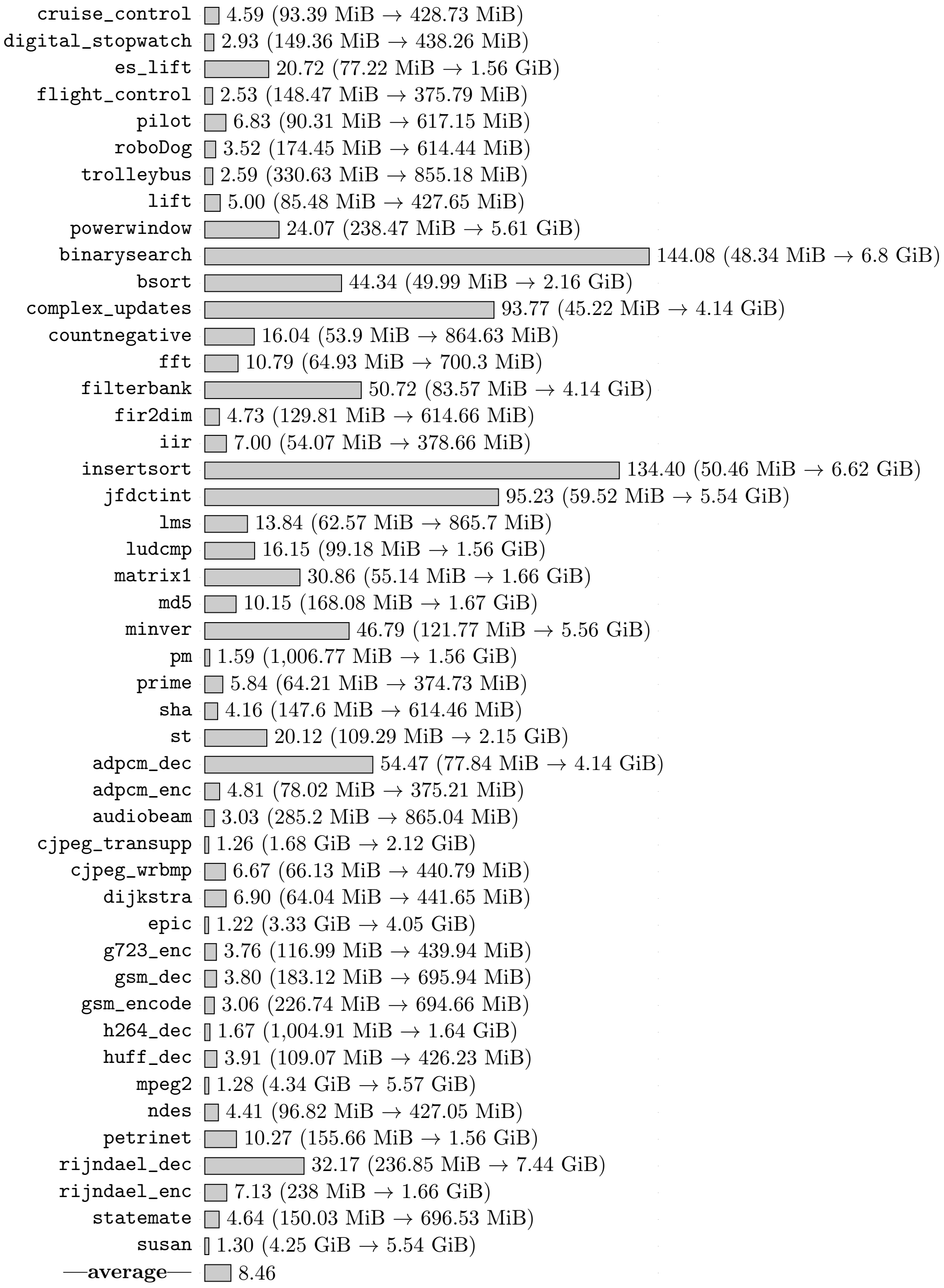


Figure B.50.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *progGran*$_{0.95}$, cf. Table 10.11) for a quad-core processor with core configuration *Conf*$_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

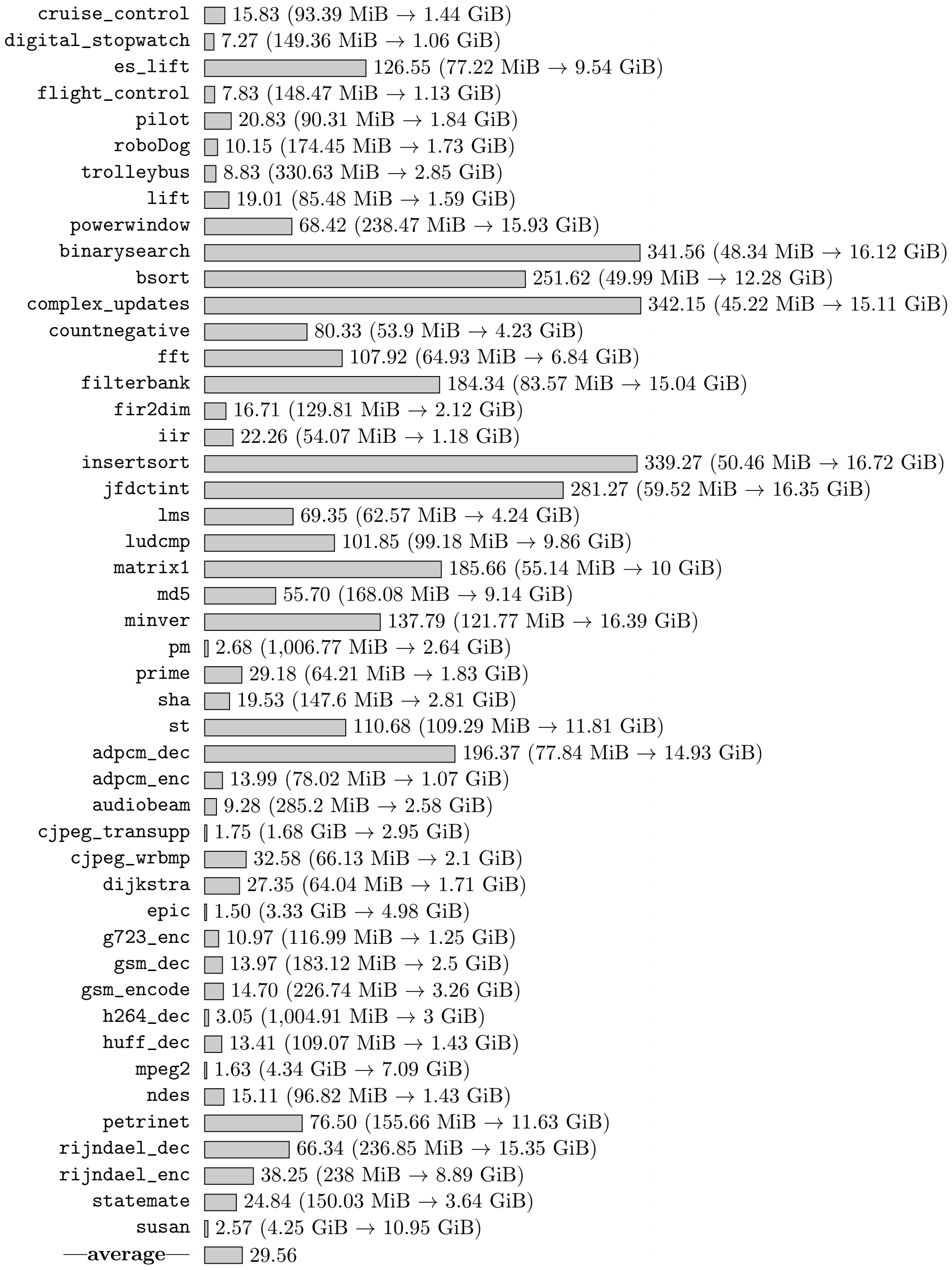


Figure B.51.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *combined*$_{0.95}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *analysis memory consumption* per benchmark *normalized* to the corresponding memory consumption of a co-runner-insensitive analysis.

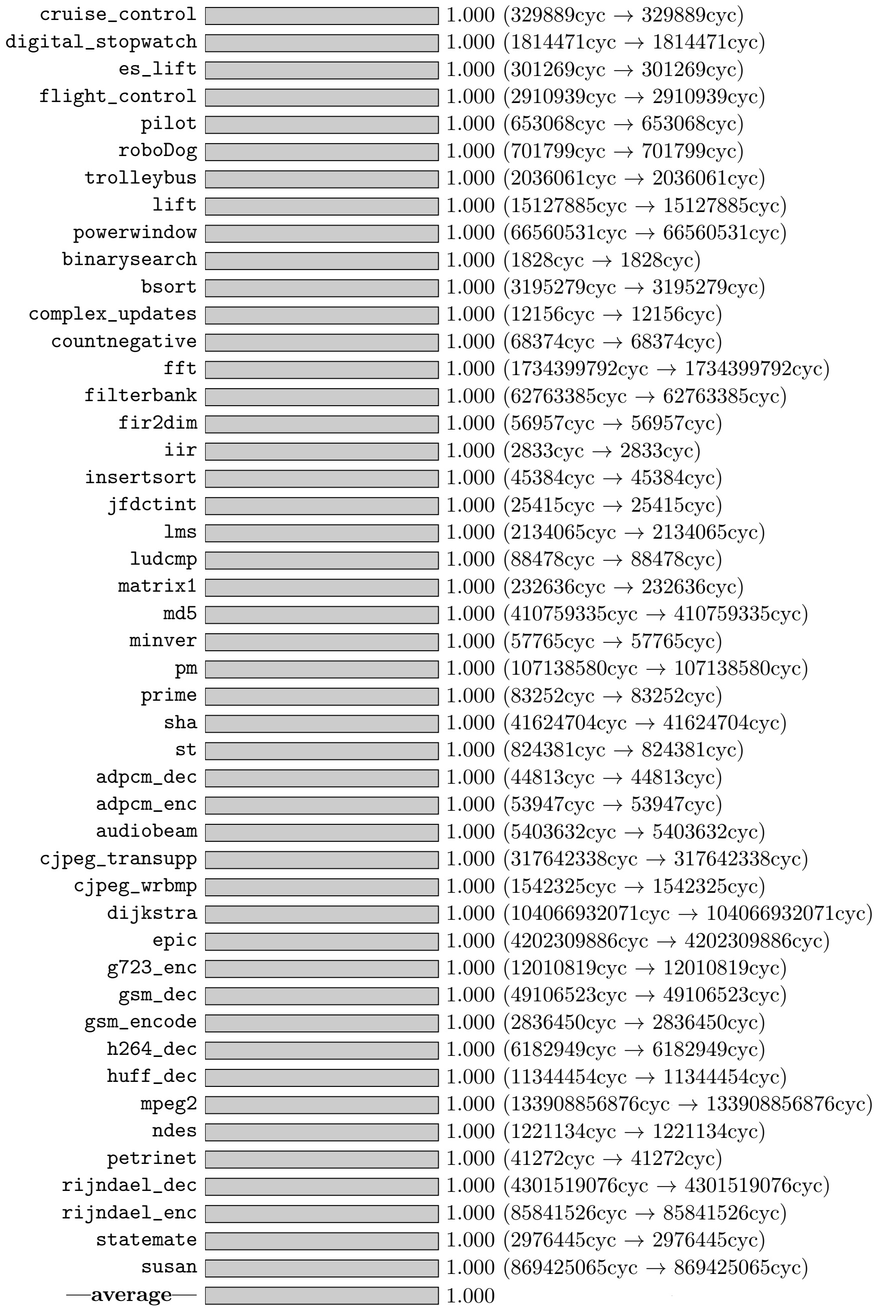


Figure B.52.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *ISPET*, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

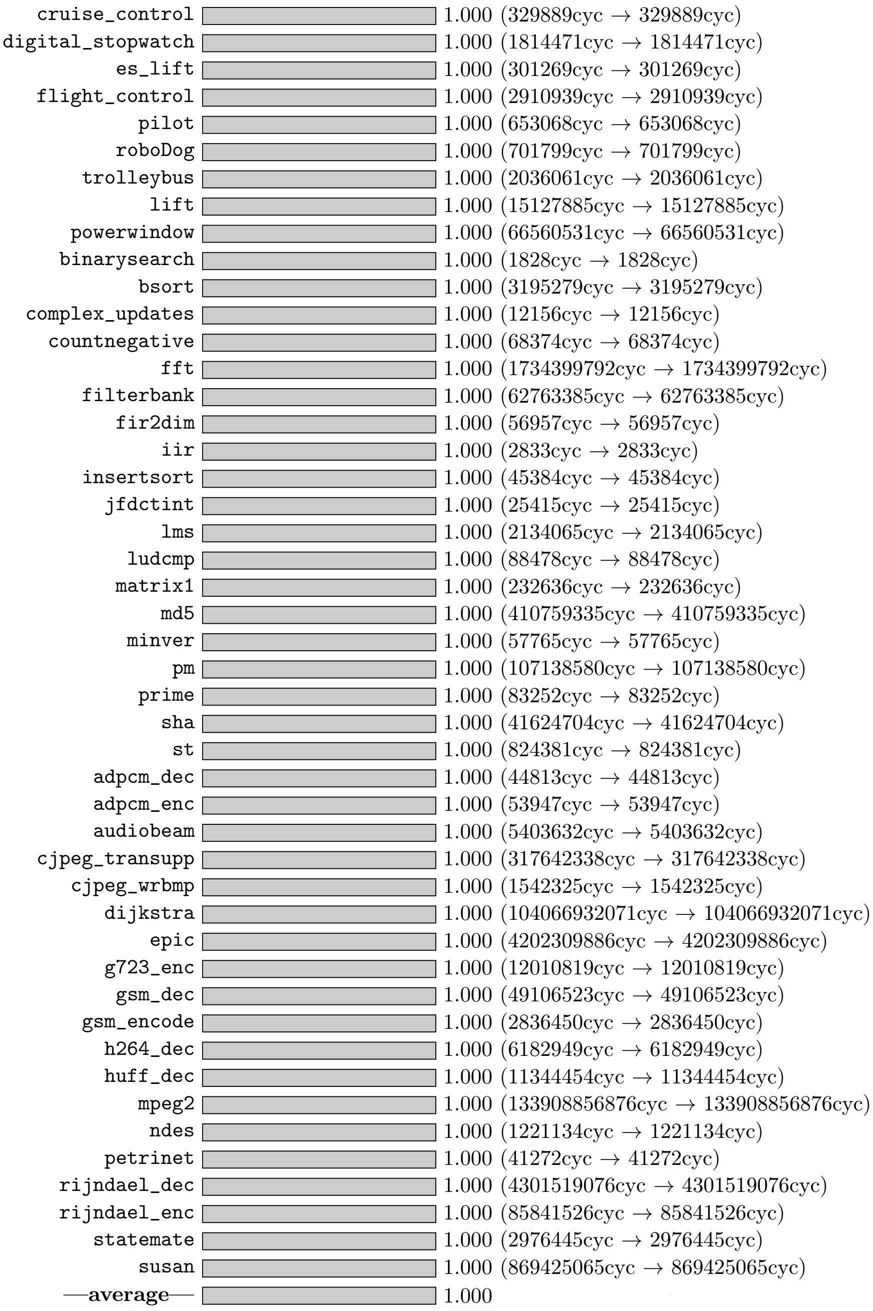


Figure B.53.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *progGran*$_{0.5}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

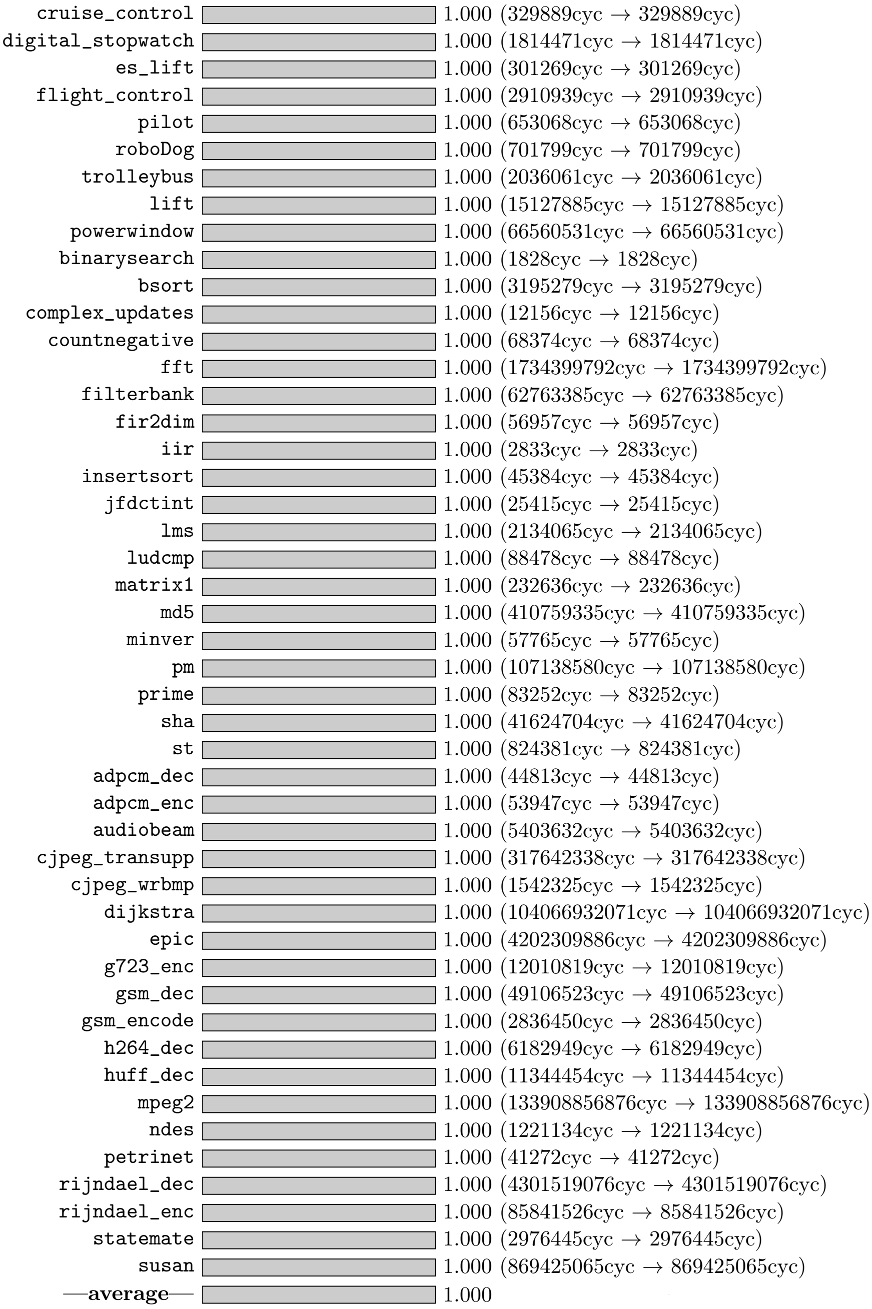


Figure B.54.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *combined*$_{0.5}$, cf. Table 10.11) for a quad-core processor with core configuration *Conf*$_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

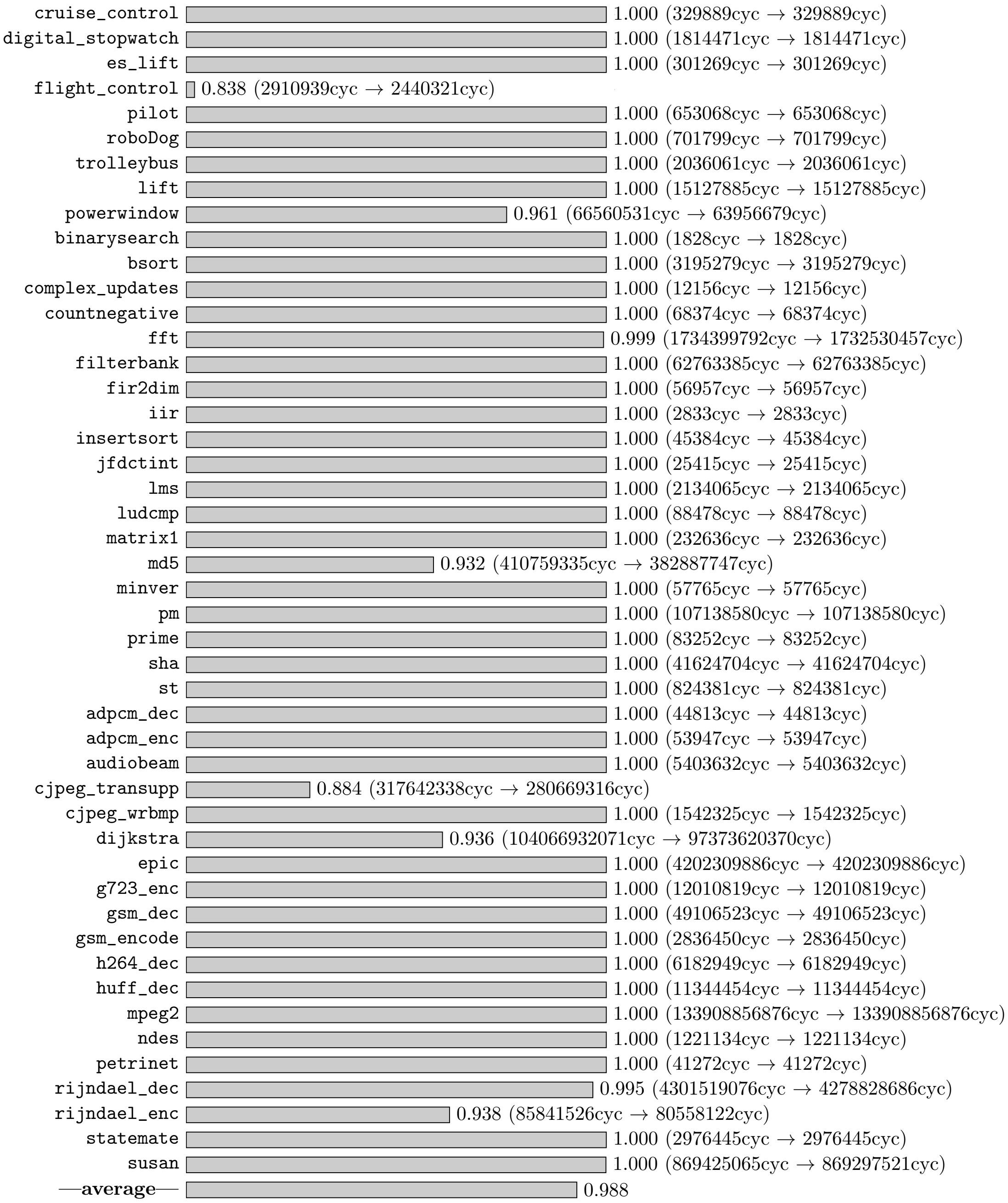


Figure B.55.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $progGran_{0.9}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

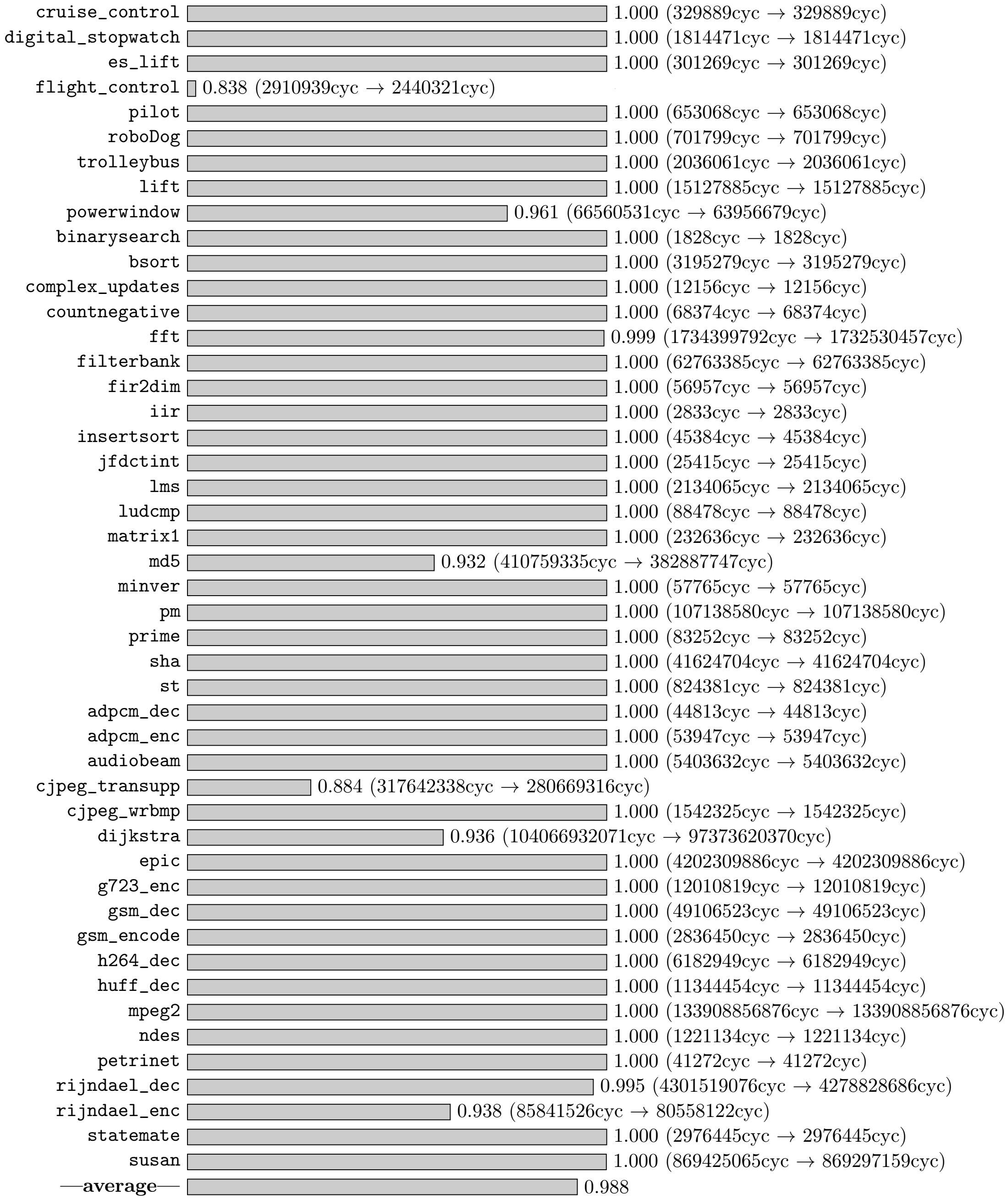


Figure B.56.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by *combined*$_{0.9}$, cf. Table 10.11) for a quad-core processor with core configuration $Conf_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

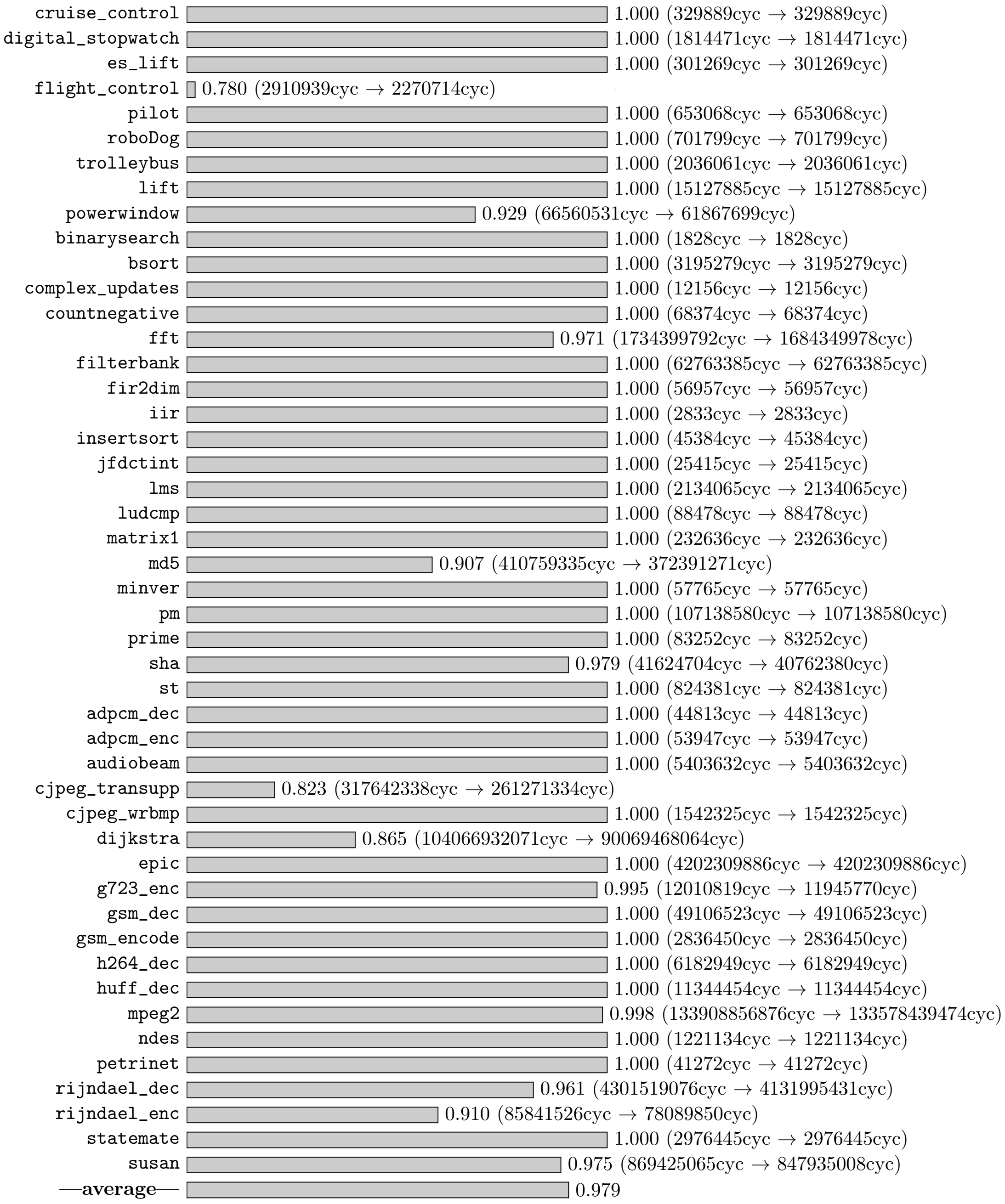


Figure B.57.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $\mathit{progGran}_{0.95}$, cf. Table 10.11) for a quad-core processor with core configuration Conf_{ic}^{ooo}: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

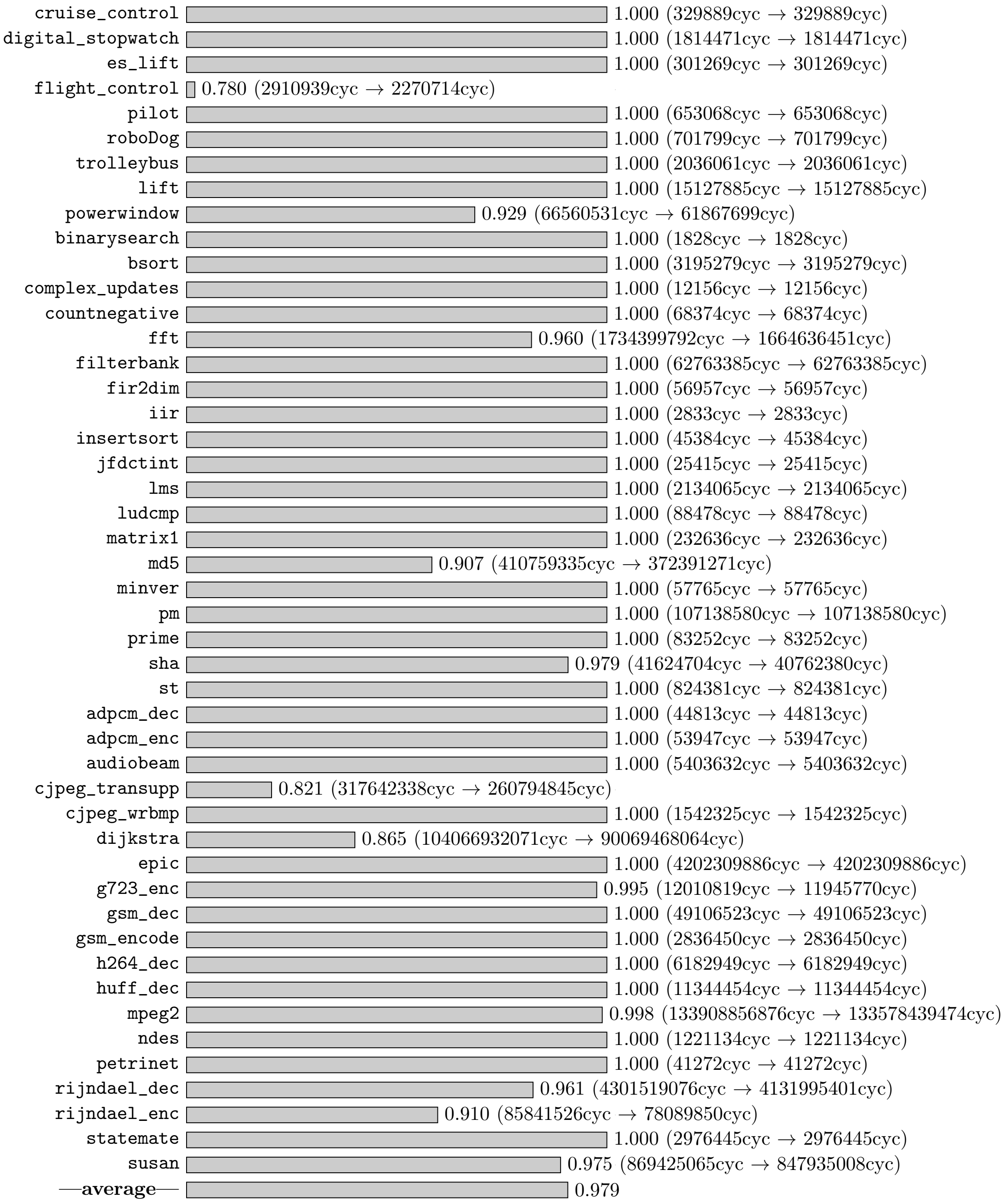


Figure B.58.: Co-runner-sensitive WCET analysis (arrival curve values calculated as defined by $\textit{combined}_{0.95}$, cf. Table 10.11) for a quad-core processor with core configuration $\textit{Conf}_{ic}^{ooo}$: *WCET bounds* per benchmark *normalized* to the corresponding WCET bounds of a co-runner-insensitive analysis.

Appendix C

Cited References

Ich will lieber stehend sterben
Als kniend leben
Lieber tausend Qualen leiden
Als einmal aufzugeben

(Lieber stehend sterben, Böhse Onkelz, 1993)

Abel, A., Benz, F., Doerfert, J., Dörr, B., Hahn, S., Haupenthal, F., Jacobs, M., Moin, A. H., Reineke, J., Schommer, B., and Wilhelm, R. (2013). Impact of resource sharing on performance and performance prediction: A survey. In *Proceedings of the 24th Conference on Concurrency Theory*, CONCUR 2013, pages 25–43. On pages 4, 8, 109, and 114.

Abel, A. and Reineke, J. (2013). Measurement-based modeling of the cache replacement policy. In *19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA, April 9-11, 2013*, pages 65–74. On page 241.

Abel, A. and Reineke, J. (2019). uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*. On page 241.

Altmeyer, S. (2013). *Analysis of preemptively scheduled hard real-time systems.* PhD thesis, Saarland University. On pages 148 and 195.

Altmeyer, S., Davis, R. I., Indrusiak, L. S., Maiza, C., Nélis, V., and Reineke, J. (2015). A generic and compositional framework for multicore response time analysis. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems*, pages 129–138. On pages 8, 9, 19, 115, 130, 134, 139, 169, 171, 173, 176, and 196.

Alur, R. and Dill, D. L. (1990). Automata for modeling real-time systems. In *Automata, Languages and Programming, 17th International Colloquium, ICALP90, Warwick University, England, July 16-20, 1990, Proceedings*, pages 322–335. On page 30.

Andersson, B., Easwaran, A., and Lee, J. (2010). Finding an upper bound on the increase in execution time due to contention on the memory bus in cots-based multicore systems. *SIGBED Review*, 7(1):4. On pages 9, 19, and 130.

Ballabriga, C., Cassé, H., Rochange, C., and Sainrat, P. (2010). OTAWA: an open toolbox for adaptive WCET analysis. In *Software Technologies for Embedded and Ubiquitous Systems - 8th IFIP WG 10.2 International Workshop, SEUS 2010, Waidhofen/Ybbs, Austria, October 13-15, 2010. Proceedings*, pages 35–46. On page 79.

Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. (1995). UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems III: Verification and Control, Proceedings of the DIMACS/SYCON Workshop, October 22-25, 1995, Ruttgers University, New Brunswick, NJ, USA*, pages 232–243. On page 9.

Blaß, T., Hahn, S., and Reineke, J. (2017). Write-back caches in WCET analysis. In *29th Euromicro Conference on Real-Time Systems, ECRTS 2017, June 27-30, 2017, Dubrovnik, Croatia*, pages 26:1–26:22. On pages 63, 64, and 142.

Blazy, S., Maroneze, A. O., and Pichardie, D. (2013). Formal verification of loop bound estimation for WCET analysis. In Cohen, E. and Rybalchenko, A., editors, *Verified Software: Theories, Tools, Experiments - 5th International Conference, VSTTE 2013, Menlo Park, CA, USA, May 17-19, 2013, Revised Selected Papers*, volume 8164 of *Lecture Notes in Computer Science*, pages 281–303. Springer. On page 64.

Bonenfant, A., Cassé, H., Michiel, M. D., Knoop, J., Kovács, L., and Zwirchmayr, J. (2012). FFX: a portable WCET annotation language. In *20th International Conference on Real-Time and Network Systems, RTNS '12, Pont a Mousson, France - November 08 - 09, 2012*, pages 91–100. On page 100.

Boudec, J. L. and Thiran, P. (2001). *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer. On pages 173 and 182.

Bui, D. N., Lee, E. A., Liu, I., Patel, H. D., and Reineke, J. (2011). Temporal isolation on multiprocessing architectures. In *Proceedings of the 48th Design Automation Conference, DAC 2011, San Diego, California, USA, June 5-10, 2011*, pages 274–279. On page 7.

Busquets-Mataix, J. V., Serrano, J. J., and Wellings, A. J. (1997). Hybrid instruction cache partitioning for preemptive real-time systems. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems, RTS 1997, 11-13 June, 1997, Toledo, Spain*, pages 56–63. On page 7.

Chattopadhyay, S., Kee, C., Roychoudhury, A., Kelter, T., Marwedel, P., and Falk, H. (2012). A unified WCET analysis framework for multi-core platforms. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 99–108. On pages 8, 9, and 30.

Chattopadhyay, S., Roychoudhury, A., and Mitra, T. (2010). Modeling shared cache and bus in multi-cores for timing analysis. In *13th International Workshop on Software and Compilers for Embedded Systems, SCOPES '10, St. Goar, Germany, June 29-30, 2010*, page 6. On pages 9 and 19.

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169. On page 31.

Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California. ACM Press, New York, NY. On pages 30, 41, 46, 62, and 237.

Cullmann, C. (2013). *Cache persistence analysis for embedded real-time systems*. PhD thesis, Saarland University. On pages 16, 63, 64, 85, 93, 96, and 142.

Dasari, D., Andersson, B., Nélis, V., Petters, S. M., Easwaran, A., and Lee, J. (2011). Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *Proceedings of the 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075. On pages 9, 19, and 130.

Dasari, D. and Nélis, V. (2012). An analysis of the impact of bus contention on the WCET in multicores. In *Proceedings of the 14th IEEE International Conference on High Performance Computing and Communication & the 9th IEEE International Conference on Embedded Software and Systems*, pages 1450–1457. On pages 9, 19, and 130.

Davis, R. I. and Navet, N. (2012). Traffic shaping to reduce jitter in controller area network (can). *SIGBED Rev.*, 9(4):37–40. On page 240.

Diefendorff, K., Oehler, R., and Hochsprung, R. (1994). Evolution of the powerpc architecture. *IEEE Micro*, 14(2):34–49. On page 68.

Diefendorff, K. and Silha, E. (1994). The powerpc user instruction set architecture. *IEEE Micro*, 14(5):30–41. On page 68.

Edwards, S. A. and Lee, E. A. (2007). The case for the precision timed (PRET) machine. In *Proceedings of the 44th Design Automation Conference, DAC 2007, San Diego, CA, USA, June 4-8, 2007*, pages 264–265. On page 7.

Engblom, J. and Ermedahl, A. (2000). Modeling complex flows for worst-case execution time analysis. In *Proceedings of the 21st IEEE Real-Time Systems Symposium (RTSS 2000), Orlando, Florida, USA, 27-30 November 2000*, pages 163–174. On pages 15, 63, 64, 91, 93, 94, 96, 100, and 142.

Ermedahl, A. and Gustafsson, J. (1997). Deriving annotations for tight calculation of execution time. In *Euro-Par '97 Parallel Processing, Third International Euro-Par Conference, Passau, Germany, August 26-29, 1997, Proceedings*, pages 1298–1307. On page 100.

Falk, H., Altmeyer, S., Hellinckx, P., Lisper, B., Puffitsch, W., Rochange, C., Schoeberl, M., Sorensen, R. B., Wägemann, P., and Wegener, S. (2016). Taclebench: A benchmark collection to support worst-case execution time research. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 2:1–2:10. On pages 100 and 148.

Faymonville, C. M. (2015). Evaluating compositional timing analyses. Master's thesis, Saarland University. On page 154.

Flanagan, C. and Qadeer, S. (2003). Thread-modular model checking. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 213–224. On pages 10, 15, 41, and 233.

Gan, Z. and Gu, Z. (2015). WCET-aware task assignment and cache partitioning for WCRT minimization on multi-core systems. In *Seventh International Symposium on Parallel Architectures, Algorithms and Programming, PAAP 2015, Nanjing, China, December 12-14, 2015*, pages 143–148. On page 7.

Georgiadis, L., Guérin, R., Peris, V., and Sivarajan, K. N. (1996). Efficient network qos provisioning based on per node traffic shaping. *IEEE/ACM Trans. Netw.*, 4(4):482–501. On page 240.

Giannopoulou, G., Lampka, K., Stoimenov, N., and Thiele, L. (2012). Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems. In *Proceedings of the 10th ACM International Conference on Embedded Software*, EMSOFT '12, pages 63–72, New York, NY, USA. ACM. On pages 9, 30, 130, and 238.

Gotsman, A., Berdine, J., Cook, B., and Sagiv, M. (2007). Thread-modular shape analysis. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 266–277. On pages 10, 41, and 233.

Graf, S. and Saïdi, H. (1997). Construction of abstract state graphs with PVS. In *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, pages 72–83. On pages 15 and 30.

Guan, N., Stigge, M., Yi, W., and Yu, G. (2009). Cache-aware scheduling and analysis for multicores. In *Proceedings of the 9th ACM & IEEE International conference on Embedded software, EMSOFT 2009, Grenoble, France, October 12-16, 2009*, pages 245–254. On page 8.

Gustavsson, A., Ermedahl, A., Lisper, B., and Pettersson, P. (2010). Towards WCET analysis of multicore architectures using UPPAAL. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, July 6, 2010, Brussels, Belgium*, pages 101–112. On pages 9, 19, and 129.

Hahn, S. (2018). *On static execution-time analysis*. PhD thesis. On page 240.

Hahn, S., Jacobs, M., and Reineke, J. (2016a). Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France, October 19-21, 2016*, pages 299–308. On pages 4, 6, 8, 9, 10, 11, 19, 20, 94, 134, 135, 138, 148, and 163.

Hahn, S., Jacobs, M., and Reineke, J. (2016b). Enabling compositionality for multicore timing analysis. Technical report, Saarland University. On page 159.

Hahn, S., Reineke, J., and Wilhelm, R. (2013). Towards compositionality in execution time analysis – definition and challenges. In *Proceedings of the 6th International Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, CRTS 2013. On pages 4, 10, 20, 31, 133, 134, 135, 153, and 172.

Hardy, D., Piquet, T., and Puaut, I. (2009). Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 68–77. On pages 9 and 19.

Healy, C. A., Sjödin, M., Rustagi, V., and Whalley, D. B. (1998). Bounding loop iterations for timing analysis. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium, RTAS'98, Denver, Colorado, USA, June 3-5, 1998*, pages 12–21. On page 100.

Hedley, D. and Hennell, M. A. (1985). The causes and effects of infeasible paths in computer programs. In *Proceedings, 8th International Conference on Software Engineering, London, UK, August 28-30, 1985.*, pages 259–267. On page 64.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition. On pages 66, 114, 115, and 147.

Henzinger, T. A., Jhala, R., Majumdar, R., and Qadeer, S. (2003). Thread-modular abstraction refinement. In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, pages 262–274. On pages 10, 41, and 233.

Higham, N. J. (2002). *Accuracy and stability of numerical algorithms (2. ed.)*. SIAM. On page 64.

Hong, Y., Huang, W., and Kuo, C. (2010). *Cooperative Communications and Networking: Technologies and System Design*. Springer US. On page 115.

Huang, W., Chen, J., and Reineke, J. (2016). MIRROR: symmetric timing analysis for real-time tasks on multicore platforms with shared resources. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, pages 158:1–158:6. On pages 9 and 19.

Jacobs, M. (2013). Improving the precision of approximations in WCET analysis for multi-core processors. In *Proceedings of the 7th Junior Researcher Workshop on Real-Time Computing*, JRWRTC 2013, pages 1–4. On pages 6, 11, and 30.

Jacobs, M., Hahn, S., and Hack, S. (2015). WCET analysis for multi-core processors with shared buses and event-driven bus arbitration. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems*, RTNS '15, pages 193–202, New York, NY, USA. ACM. On pages 6, 8, 11, 79, 93, 119, 124, 148, 150, 153, 169, 171, and 239.

Jacobs, M., Hahn, S., and Hack, S. (2016). A framework for the derivation of WCET analyses for multi-core processors. In *Proceedings of the 28th Euromicro Conference on Real-Time Systems*, pages 141–151. On pages 6 and 11.

John, M. and Jacobs, M. (2014). A framework for the optimization of the WCET of programs on multi-core processors. In *Proceedings of the 8th Junior Researcher Workshop on Real-Time Computing*, JRWRTC 2014, pages 1–4. On pages 7 and 240.

Karp, R. M. (1972). Reducibility among combinatorial problems. In *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.*, pages 85–103. On pages 17, 65, and 86.

Kästner, D. and Wilhelm, S. (2002). Generic control flow reconstruction from assembly code. In *Proceedings of the 2002 Joint Conference on Languages, Compilers, and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02), Berlin, Germany, 19-21 June 2002*, pages 46–55. On page 79.

Kelter, T. (2015). *WCET analysis and optimization for multi-core real-time systems*. PhD thesis, Technical University Dortmund, Germany. On pages 9, 19, and 238.

Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., and Roychoudhury, A. (2011). Bus-aware multicore WCET analysis through TDMA offset bounds. In *23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011*, pages 3–12. On page 8.

Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., and Roychoudhury, A. (2014). Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, 50(2):185–229. On page 8.

Kelter, T. and Marwedel, P. (2014). Parallelism analysis: Precise WCET values for complex multi-core systems. In *Revised Selected Papers of the 3rd International Workshop on Formal Techniques for Safety-Critical Systems*, pages 142–158. On pages 9, 10, 19, 30, and 130.

Kirner, R., Knoop, J., Prantl, A., Schordan, M., and Kadlec, A. (2011). Beyond loop bounds: comparing annotation languages for worst-case execution time analysis. *Software and System Modeling*, 10(3):411–437. On page 100.

Kovalev, M., Müller, S. M., and Paul, W. J. (2014). *A Pipelined Multi-core MIPS Machine - Hardware Implementation and Correctness Proof*, volume 9000 of *Lecture Notes in Computer Science*. Springer. On page 115.

Lampka, K., Giannopoulou, G., Pellizzoni, R., Wu, Z., and Stoimenov, N. (2014). A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets. *Real-Time Systems*, 50(5-6):736–773. On pages 9, 130, and 238.

Li, Y., Suhendra, V., Liang, Y., Mitra, T., and Roychoudhury, A. (2009). Timing analysis of concurrent programs running on shared cache multi-cores. In *Proceedings of the 30th IEEE Real-Time Systems Symposium, RTSS 2009, Washington, DC, USA, 1-4 December 2009*, pages 57–67. On pages 9, 19, and 133.

Li, Y. S., Malik, S., and Wolfe, A. (1995). Performance estimation of embedded software with instruction cache modeling. In *Proceedings of the 1995 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1995, San Jose, California, USA, November 5-9, 1995*, pages 380–387. On pages 15, 63, 64, 94, 95, and 206.

Li, Y. S., Malik, S., and Wolfe, A. (1996). Cache modeling for real-time software: beyond direct mapped instruction caches. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96), December 4-6, 1996, Washington, DC, USA*, pages 254–263. On pages 9, 15, 63, 64, 94, 95, and 206.

Li, Y.-T. S. and Malik, S. (1995). Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA. ACM. On pages 15, 58, 62, 63, 64, 91, 93, 94, 96, 100, 104, 105, 106, 142, 164, 166, 182, 205, and 237.

Liang, Y., Ding, H., Mitra, T., Roychoudhury, A., Li, Y., and Suhendra, V. (2012). Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48:638–680. On page 40.

Liu, I., Reineke, J., Broman, D., Zimmer, M., and Lee, E. A. (2012). A PRET microarchitecture implementation with repeatable timing and competitive performance. In *30th International IEEE Conference on Computer Design, ICCD 2012, Montreal, QC, Canada, September 30 - Oct. 3, 2012*, pages 87–93. On page 7.

Liu, T., Zhao, Y., Li, M., and Xue, C. J. (2010). Task assignment with cache partitioning and locking for WCET minimization on MPSoC. In *39th International Conference on Parallel Processing, ICPP 2010, San Diego, California, USA, 13-16 September 2010*, pages 573–582. On pages 7 and 8.

Liu, T., Zhao, Y., Li, M., and Xue, C. J. (2011). Joint task assignment and cache partitioning with cache locking for WCET minimization on mpsoc. *J. Parallel Distrib. Comput.*, 71(11):1473–1483. On page 7.

Lundqvist, T. and Stenstrom, P. (1999). Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 12–21. On pages 4, 10, 20, 94, 134, 135, and 172.

Lv, M., Guan, N., Reineke, J., Wilhelm, R., and Yi, W. (2016). A survey on static cache analysis for real-time systems. *LITES*, 3(1):05:1–05:48. On pages 9 and 206.

Lv, M., Yi, W., Guan, N., and Yu, G. (2010). Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA, November 30 - December 3, 2010*, pages 339–349. On pages 8, 9, and 129.

Maksoud, M. A. (2015). *Processor pipelines in WCET analysis*. PhD thesis, Saarland University. On page 93.

Maksoud, M. A. and Reineke, J. (2014). A compiler optimization to increase the efficiency of WCET analysis. In *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, page 87. On page 93.

Malkis, A., Podelski, A., and Rybalchenko, A. (2006). Thread-modular verification is cartesian abstract interpretation. In *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium, Tunis, Tunisia, November 20-24, 2006, Proceedings*, pages 183–197. On page 41.

Malkis, A., Podelski, A., and Rybalchenko, A. (2007). Precise thread-modular verification. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, pages 218–232. On pages 10, 41, and 233.

Maroneze, A. O., Blazy, S., Pichardie, D., and Puaut, I. (2014). A formally verified WCET estimation tool. In Falk, H., editor, *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, volume 39 of *OASICS*, pages 11–20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. On page 64.

Matthies, N. (2006). *Präzise Bestimmung längster Programmpfade anhand von Zustandsgraphen unter Berücksichtigung von Schleifen-Nebenbedingungen*. Diplomarbeit, Saarland University. On pages 15, 62, and 85.

Mauborgne, L. and Rival, X. (2005). Trace partitioning in abstract interpretation based static analyzers. In Sagiv, M., editor, *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer Berlin Heidelberg. On page 46.

Miné, A. (2011). Static analysis of run-time errors in embedded critical parallel C programs. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 398–418. On pages 10, 41, and 233.

Miné, A. (2014). Relational thread-modular static value analysis by abstract interpretation. In *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, pages 39–58. On pages 10, 41, and 233.

Monat, R. and Miné, A. (2017). Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings*, pages 386–404. On pages 10, 41, and 233.

Mussot, V., Ruiz, J., Sotin, P., Michiel, M. D., and Cassé, H. (2016). Expressing and exploiting conflicts over paths in WCET analysis. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, pages 3:1–3:11. On page 103.

Mussot, V. and Sotin, P. (2015). Improving WCET analysis precision through automata product. In *21st IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2015, Hong Kong, China, August 19-21, 2015*, pages 207–216. On pages 64, 101, 104, and 105.

Nagar, K. (2016). *Precise Analysis of Private and Shared Caches for Tight WCET Estimates.* PhD thesis, Indian Institute of Science, Bangalore. On page 172.

Nagar, K. and Srikant, Y. (2014). Precise shared cache analysis using optimal interference placement. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*, pages 125–134. On pages 9, 19, 94, and 133.

Nagar, K. and Srikant, Y. N. (2015). Path sensitive cache analysis using cache miss paths. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 43–60. On pages 63, 64, and 142.

Nagar, K. and Srikant, Y. N. (2016). Fast and precise worst-case interference placement for shared cache analysis. *ACM Trans. Embedded Comput. Syst.*, 15(3):45:1–45:26. On pages 9, 19, and 94.

Negrean, M., Schliecker, S., and Ernst, R. (2009). Response-time analysis of arbitrarily activated tasks in multiprocessor systems with shared resources. In *Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009*, pages 524–529. On pages 9, 19, and 130.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of Program Analysis.* Springer-Verlag New York, Inc., Secaucus, NJ, USA. On pages 46, 250, and 251.

Nowotsch, J. (2014). *Interference-sensitive Worst-case Execution Time Analysis for Multi-core Processors.* PhD thesis, University of Augsburg. On pages 8, 9, and 19.

Nowotsch, J. and Paulitsch, M. (2013). Quality of service capabilities for hard real-time applications on multi-core processors. In *21st International Conference on Real-Time Networks and Systems, RTNS 2013, Sophia Antipolis, France, October 17-18, 2013*, pages 151–160. On pages 9, 19, and 130.

Nowotsch, J., Paulitsch, M., Henrichsen, A., Pongratz, W., and Schacht, A. (2014). Monitoring and WCET analysis in COTS multi-core-soc-based mixed-criticality systems. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, pages 1–5. On pages 9, 19, and 130.

Oehlert, D., Saidi, S., and Falk, H. (2019). Code-inherent traffic shaping for hard real-time systems. *ACM Trans. Embed. Comput. Syst.*, 18(5s). On page 240.

Paolieri, M., Quiñones, E., Cazorla, F. J., Bernat, G., and Valero, M. (2009). Hardware support for WCET analysis of hard real-time multicore systems. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, pages 57–68. On page 8.

Pellizzoni, R., Schranzhofer, A., Chen, J.-J., Caccamo, M., and Thiele, L. (2010). Worst case delay analysis for memory interference in multicore systems. In *Proceedings of the 13th Conference on Design, Automation and Test in Europe*, DATE '10, pages 741–746, 3001 Leuven, Belgium, Belgium. European Design and Automation Association. On pages 9, 19, 115, 130, and 179.

Perret, Q., Maurère, P., Noulard, E., Pagetti, C., Sainrat, P., and Triquet, B. (2016). Temporal isolation of hard real-time applications on many-core processors. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, pages 37–47. On page 7.

Plazar, S., Lokuciejewski, P., and Marwedel, P. (2009). WCET-aware software based cache partitioning for multi-task real-time systems. In *9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009.* On page 8.

Puaut, I. and Hardy, D. (2014). Heptane static WCET estimation tool. `https://team.inria.fr/alf/software/heptane`. Accessed: 2016-11-16. On page 79.

Puschner, P. (2003). The single-path approach towards WCET-analysable software. In *Proceedings of the IEEE International Conference on Industrial Technology*, volume 2, pages 699–704. On page 179.

Puschner, P. P. and Schedl, A. V. (1997). Computing maximum task execution times - A graph-based approach. *Real-Time Systems*, 13(1):67–91. On pages 15, 61, 62, 64, 164, 166, and 237.

Raymond, P. (2014). A general approach for expressing infeasibility in implicit path enumeration technique. In *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, pages 8:1–8:9. On pages 63, 64, 100, 103, 105, and 142.

Rihani, H., Moy, M., Maiza, C., and Altmeyer, S. (2015). WCET analysis in shared resources real-time systems with TDMA buses. In *Proceedings of the 23rd International Conference on Real Time Networks and Systems, RTNS 2015, Lille, France, November 4-6, 2015*, pages 183–192. On page 8.

Rosen, J., Andrei, A., Eles, P., and Peng, Z. (2007). Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 49–60, Washington, DC, USA. IEEE Computer Society. On pages 7 and 240.

Ruiz, J. and Cassé, H. (2015). Using SMT solving for the lookup of infeasible paths in binary programs. In *15th International Workshop on Worst-Case Execution Time Analysis, WCET 2015, July 7, 2015, Lund, Sweden*, pages 95–104. On page 100.

Sagiv, S., Reps, T. W., and Wilhelm, R. (2002). Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298. On pages 15 and 30.

Satpathy, S., Das, R., Dreslinski, R. G., Mudge, T. N., Sylvester, D., and Blaauw, D. (2012). High radix self-arbitrating switch fabric with multiple arbitration schemes and quality of service. In *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 406–411. On page 115.

Schliecker, S. and Ernst, R. (2010). Real-time performance analysis of multiprocessor systems with shared memory. *ACM Trans. Embedded Comput. Syst.*, 10(2):22. On pages 9, 19, 130, 171, 173, 176, and 196.

Schliecker, S., Negrean, M., and Ernst, R. (2009). Response time analysis in multicore ECUs with shared resources. *IEEE Trans. Industrial Informatics*, 5(4):402–413. On pages 9 and 19.

Schliecker, S., Negrean, M., Nicolescu, G., Paulin, P. G., and Ernst, R. (2008). Reliable performance analysis of a multicore multithreaded system-on-chip. In *Proceedings of the 6th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 161–166. On pages 9, 19, 130, and 139.

Schranzhofer, A., Chen, J.-J., and Thiele, L. (2010a). Timing analysis for TDMA arbitration in resource sharing systems. In *Proceedings of the 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 215–224, Washington, DC, USA. IEEE Computer Society. On page 8.

Schranzhofer, A., Pellizzoni, R., Chen, J., Thiele, L., and Caccamo, M. (2010b). Worst-case response time analysis of resource access models in multi-core systems. In *Proceedings of the 47th Design Automation Conference, DAC 2010, Anaheim, California, USA, July 13-18, 2010*, pages 332–337. On pages 9 and 19.

Schranzhofer, A., Pellizzoni, R., Chen, J.-J., Thiele, L., and Caccamo, M. (2011). Timing analysis for resource access interference on adaptive resource arbiters. In *Proceedings of the 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '11, pages 213–222, Washington, DC, USA. IEEE Computer Society. On pages 8, 9, 19, 30, and 130.

Stein, I. and Martin, F. (2007). Analysis of path exclusion at the machine code level. In *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Pisa, Italy, July 3, 2007*. On page 100.

Stein, I. J. (2010). *ILP-based path analysis on abstract pipeline state graphs*. PhD thesis, Universität des Saarlandes, Postfach 151141, 66041 Saarbrücken. On pages 15, 16, 30, 58, 62, 63, 64, 85, 96, 102, 103, 142, and 237.

Suhendra, V. and Mitra, T. (2008). Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th Design Automation Conference, DAC 2008, Anaheim, CA, USA, June 8-13, 2008*, pages 300–303. On page 8.

Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309. On pages 250 and 251.

Theiling, H. (2000). Extracting safe and precise control flow from binaries. In *7th International Workshop on Real-Time Computing and Applications Symposium (RTCSA 2000), 12-14 December 2000, Cheju Island, South Korea*, pages 23–30. On page 79.

Theiling, H. (2002). ILP-based interprocedural path analysis. In *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, pages 349–363. On pages 15, 62, 64, and 93.

Theiling, H., Ferdinand, C., and Wilhelm, R. (2000). Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, 18(2/3):157–179. On page 93.

Thesing, S. (2004). *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarländische Universitäts- und Landesbibliothek, Postfach 151141, 66041 Saarbrücken. On pages 15, 19, 30, 46, 62, 66, 79, 80, 111, and 237.

Ungerer, T., Cazorla, F., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quinones, E., Gerdes, M., Paolieri, M., Wolf, J., Casse, H., Uhrig, S., Guliashvili, I., Houston, M., Kluge, F., Metzlaff, S., and Mische, J. (2010). Merasa: Multicore execution of hard real-time applications supporting analyzability. *IEEE Micro*, 30:66–75. On page 8.

Wegener, S. (2017). Towards multicore WCET analysis. In *17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, June 27, 2017, Dubrovnik, Croatia*, pages 7:1–7:12. On pages 9, 19, and 163.

Wieder, A. and Brandenburg, B. B. (2013). On spin locks in AUTOSAR: blocking analysis of fifo, unordered, and priority-ordered spin locks. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, pages 45–56. On page 30.

Yan, J. and Zhang, W. (2008). WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, April 22-24, 2008, St. Louis, Missouri, USA*, pages 80–89. On pages 8, 9, 19, and 133.

Zang, W. and Gordon-Ross, A. (2016). CaPPS: cache partitioning with partial sharing for multi-core embedded systems. *Design Automation for Embedded Systems*, 20(1):65–92. On page 8.

Zhang, W. and Yan, J. (2009). Accurately estimating worst-case execution time for multi-core processors with shared direct-mapped instruction caches. In *15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009, Beijing, China, 24-26 August 2009*, pages 455–463. On pages 9 and 133.

Fuck you, I won't do what you tell me.

(Killing in the Name, Rage Against the Machine, 1992)