Dissertation

zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
an der Fakultät für Mathematik und Informatik
der Universität des Saarlandes

On Static Execution-Time Analysis

Compositionality, Pipeline Abstraction, and Predictable Hardware

eingereicht von
Sebastian Hahn

Saarbrücken, 2018

UNIVERSITÄT
DES
SAARLANDES

Tag des Kolloquiums

4. April 2019

Dekan

Prof. Dr. Sebastian Hack

Prüfungsausschuss

Prof. Dr. Sebastian Hack	(Vorsitzender)
Prof. Dr. Jan Reineke	(Berichterstatter)
Prof. Dr. Reinhard Wilhelm	(Berichterstatter)
Prof. Dr. Bernd Becker	(Berichterstatter)
Dr. Roland Leißa	(akademischer Mitarbeiter)

Abstract

Proving timeliness is an integral part of the verification of safety-critical real-time systems. To this end, timing analysis computes upper bounds on the execution times of programs that execute on a given hardware platform. Modern hardware platforms commonly exhibit counter-intuitive timing behaviour: a locally slower execution can lead to a faster overall execution. Such behaviour challenges efficient timing analysis.

In this work, we present and discuss a hardware design, the *strictly in-order pipeline*, that behaves monotonically w.r.t. the progress of a program's execution. Based on monotonicity, we prove the absence of the aforementioned counter-intuitive behaviour.

At least since multi-core processors have emerged, timing analysis separates concerns by analysing different aspects of the system's timing behaviour individually. In this work, we validate the underlying assumption that a timing bound can be soundly composed from individual contributions. We show that even simple processors exhibit counter-intuitive behaviour—a locally slow execution can lead to an even slower overall execution—that impedes the soundness of the composition. We present the *compositional base bound* analysis that accounts for any such amplifying effects within its timing contribution. This enables a sound compositional analysis even for complex processors. Furthermore, we discuss hardware modifications that enable efficient compositional analyses.

Zusammenfassung

Echtzeitsysteme müssen unter allen Umständen beweisbar pünktlich arbeiten. Zum Beweis errechnet die Zeitanalyse obere Schranken der für die Ausführung von Programmen auf einer Hardware-Plattform benötigten Zeit. Moderne Hardware-Plattformen sind bekannt für unerwartetes Zeitverhalten bei dem eine lokale Verzögerung in einer global schnelleren Ausführung resultiert. Solches Zeitverhalten erschwert eine effiziente Analyse.

Im Rahmen dieser Arbeit diskutieren wir das Design eines Prozessors mit eingeschränkter Fließbandverarbeitung (*strictly in-order pipeline*), der sich bzgl. des Fortschritts einer Programmausführung monoton verhält. Wir beweisen, dass Monotonie das oben genannte unerwartete Zeitverhalten verhindert.

Spätestens seit dem Einsatz von Mehrkernprozessoren besteht die Zeitanalyse aus einzelnen Teilanalysen welche nur bestimmte Aspekte des Zeitverhaltens betrachten. Eine zentrale Annahme ist hierbei, dass sich die Teilergebnisse zu einer korrekten Zeitschranke zusammensetzen lassen. Im Rahmen dieser Arbeit zeigen wir, dass diese Annahme selbst für einfache Prozessoren ungültig ist, da eine lokale Verzögerung zu einer noch größeren globalen Verzögerung führen kann. Für bestehende Prozessoren entwickeln wir eine neuartige Teilanalyse, die solche verstärkenden Effekte berücksichtigt und somit eine korrekte Komposition von Teilergebnissen erlaubt. Für zukünftige Prozessoren beschreiben wir Modifikationen, die eine deutlich effizientere Zeitanalyse ermöglichen.

Acknowledgements

First of all, I thank Prof. Jan Reineke and Prof. Reinhard Wilhelm for the opportunity to pursue my research in the field of timing analysis. They provided me with inspiring ideas whenever needed and gave me the time and freedom to work out my own ideas.

I would like to thank Prof. Bernd Becker for his willingness to review my dissertation and to be part of my thesis committee.

To my dear friend and former colleague Michael Jacobs: I could not have written this thesis without you! Your steady questioning of what seemed obvious to me at first glance lead to the level of understanding that underlies this work. I really enjoyed working, travelling, and chatting about anything and everything with you.

I owe special thanks to Fabian Ritter, Barbara Dörr, Tobias Blaß, and Florian Haupenthal for proofreading the thesis. Your comments improved this thesis so much. All remaining errors are only my fault.

I would like to thank my former and current colleagues from the Compiler Design Lab and the Real-Time and Embedded Systems Lab for making the daily work pleasant; in particular Fabian, Tomasz Dudziak, Michael, Florian, and Christoph Mallon. Thank you for all the fun! I specifically enjoyed the weekly tea meetings with Florian and Christoph along with many technical and non-technical discussions. I will always have a cup of tea ready for you.

Special thanks to Christoph for extensive help with git, C/C++, and especially your contributions to the Verilog prototype of the in-order pipeline.

Our low-level timing analysis tool LLVMTA used in this thesis is the result of many people's work. Primarily, thank you, Claus Faymonville, for your major and essential contributions in the very early stage of the tool. In addition, I would like to thank all students that contributed new analyses: Tina Jung, Tobias, and Darshit Shah.

Thank you, Sandra Neumann, for organising the administrative parts of my work at the university.

Last but not least, I thank my entire family for their love and support, in particular my parents Ursula and Thomas, and my sister Bernadette.

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS), the PEP Project, 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

Chapter 1

Introduction

Real-time systems have to not only produce computationally correct results but also have to deliver their results *within a certain time*. A violation of these timing requirements can have consequences of different severity. Possible consequences range from degraded quality of service, e.g. lag in a video stream, to complete system failure, e.g. causing harm to humans. In the latter case, not a single violation of the timing requirements is acceptable. Such a system is named *hard* real-time system.

Hard real-time systems are often found embedded into physical objects that they control. Examples include the flight control system in an airplane and the airbag controller in a car. Figure 1.1 shows the general structure of an embedded control system using the example of an airbag controller. The environment of the car is sensed by measuring relevant properties such as the acceleration of the car and the distances to surrounding objects. With the sensor readings as input, a controller gauges the situation and decides on appropriate reactions, e.g. to inflate the airbag. Actuators such as the inflator take action according to the controller's output to manipulate the physical environment.

Besides the delay incurred by the sensors and actuators, the latency of the controller is crucial for satisfying the hard timing requirements. As a consequence, a hard real-time controller has to undergo a timing verification process prior to the deployment of the system. In the timing verification process, the controller must be proven to adhere to the timing requirements in all cases including the worst possible case. This thesis contributes to state-of-the-art techniques for the verification of the timing behaviour of such systems.

Nowadays, a controller is usually implemented in software running on a particular hardware platform. Thus, timing verification is a software

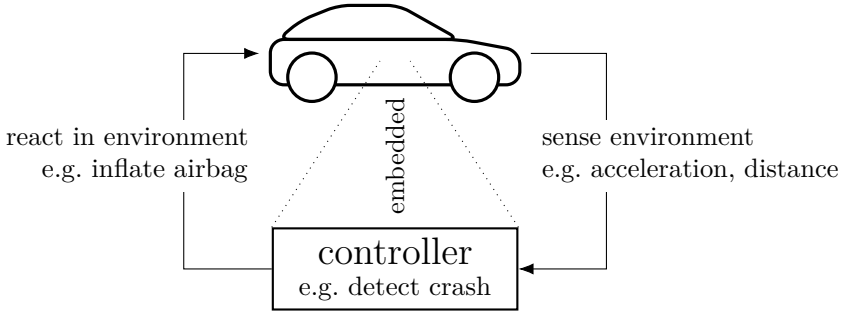


Figure 1.1: A car as an example of an embedded control system with hard real-time requirements.

verification problem. Unlike the functional behaviour of a program that is determined by the state of the program itself and its inputs, the timing behaviour of a program additionally depends on the hardware state and the concurrently executing programs. As a consequence, timing verification has to take the hardware and the execution environment into account as well.

To verify the adherence to the timing requirements, *timing analysis* derives bounds on the possible execution times of a given program as illustrated in Figure 1.2. Each technique for timing analysis

- must give *sound* results. The derived bounds are never exceeded for any input, any initial hardware state, and any concurrently executing programs.
- should give *precise* results. In particular, the overestimation, i.e. the minimal distance between an actual execution time and the derived upper bound, should be low in order to avoid an over-provisioning of system resources.
- should be *efficient* w.r.t. the available computational resources. In general, there is a trade-off between the efficiency of the analysis and the precision of the obtained results.

In theory, it is conceivable to enumerate all possible execution behaviours of a (machine) program as the actual system has finitely many states. However, such an explicit enumeration is practically infeasible due to the

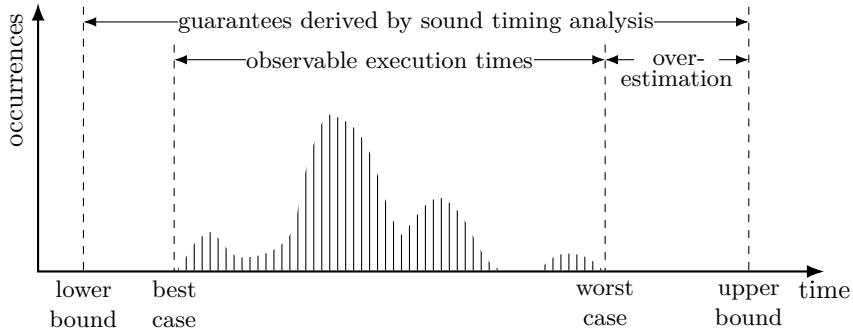


Figure 1.2: Hypothetical histogram of a program's possible execution times and basic notions in timing verification.

enormous size of the (finite) state space. *Measurement-based* approaches to timing analysis execute the given program with various inputs, in different initial hardware states, and with different concurrently executing programs. The timing estimate is then derived from the various observed execution times. As a consequence of the size of the state space, measurement-based approaches can never reach full coverage of the possible execution behaviours and thus cannot deliver provably sound bounds in general. In this thesis, we use *static* approaches to timing verification: We employ abstractions to derive guarantees about all concrete execution behaviours without the need to actually execute the program. If we base the abstractions on formal models of the concrete system, we obtain provably sound timing bounds.

Static timing analysis is commonly performed in two phases following the principle of separation of concerns. First, a low-level analysis characterises a program's behaviour when executed in isolation, i.e. without concurrently executing programs. The low-level analysis accounts for the effects of the inputs and the initial hardware states on the execution behaviour. This first phase is also known as worst-case execution time (WCET) analysis [Wilhelm et al., 2008] since the upper bound on a program's execution time has originally been the only program characterisation. Second, a high-level analysis incorporates the effects of the concurrently executing programs into a program's timing bound. To this end, the high-level analysis uses the characteristics of all programs in the system, as derived by the preceding low-level analysis.

Compositionality

The soundness of this two-phase approach to timing analysis relies on the assumption that an upper timing bound can be composed from characteristics that capture the programs' behaviour when run in isolation. This assumption that links the low-level and the high-level analysis, is commonly referred to as *timing compositionality*. Compositionality is important for a sound and efficient two-phase analysis of modern systems that feature preemptive scheduling or multi-core processors. The impact of preemptions and shared resource interference on the execution time of a program depends on the actual program schedule and can thus only be determined within the high-level analysis. Compositionality allows to account for this schedule-dependent impact on the execution time using solely the program characteristics derived by the low-level analysis. Despite its key role to overall soundness, compositionality is usually just taken for granted in the existing literature. As a consequence, it has only attracted little attention in research.

In this thesis, we demonstrate that naively composing per-program characteristics—as it is usually done in the literature—is *not* sound: it might *underestimate* the timing in the worst case. The concurrent execution of programs on a single hardware platform results in competition for the exclusive access to shared resources such as the processor or the memory. Due to this competition, a program's execution might be blocked if a required resource is already occupied by another program. The high-level analyses found in the literature take such *direct* blocking effects into account. Besides these direct effects, the execution of concurrent programs also has an impact on the state of the hardware. As a consequence, a program might encounter hardware states during its execution that could not arise when executed in isolation and that are thus not considered by the low-level analysis. These hardware states might however trigger a worse timing behaviour than the states that arise during an isolated execution. Such *indirect* effects are either not or only incompletely considered in the literature.

In this work, we make the following contributions in the context of timing compositionality. First, we show that indirect effects occur unexpectedly even on simple hardware platforms. Second, we present and discuss hardware mechanisms that provably prevent or at least limit indirect effects. Last but not least, we propose a new low-level analysis technique termed *compositional base bound* that accounts for all possible indirect effects

caused by any concurrently executing programs. For the first time, the compositional base bound enables a *sound* two-phase timing analysis with no restrictions on the hardware platform to analyse.

Towards Pipeline Abstraction

The low-level analysis is the most demanding part of timing analysis w.r.t. computational resources. It needs to consider all initial hardware states and program inputs while avoiding to explicitly enumerate all possibilities.

Abstraction allows to approximate a set of concrete elements, e.g. hardware states, by a more compact abstract element that only keeps relevant properties of the concrete elements. The problem is to find an abstraction such that the abstract elements are (a) compact enough to allow for an efficient analysis *and* (b) precise enough to derive meaningful results such as timing bounds that fulfil the timing requirements. For some parts of the state space, such abstractions are known, e.g. the interval abstraction to represent possible (input) values or the must- and may-cache abstraction for cache memories. However, no abstractions have been found for the control part of processor pipelines. As a consequence, state-of-the-art low-level analyses explore *concrete* pipeline control states that arise during program execution.

In this thesis, we examine a novel idea for an abstraction based on the notion of *progress* of a program's execution in a processor pipeline. An abstract pipeline state hereby represents all concrete pipeline states in which the program's execution has progressed *at least* as much. We provide the necessary formal background to reason about the soundness of the progress-based abstraction.

We show how to instantiate the progress-based abstraction for the subcategory of processor pipelines that behave *monotonically*: If a program's execution in one pipeline state has progressed at least as much as in another one, this order is preserved after processing an additional clock cycle.

We present a new pipeline specifically designed with monotonicity in mind: the *strictly in-order pipeline*. In contrast to a conventional in-order pipeline, the strictly in-order pipeline additionally ensures that all—instruction and data—accesses on the memory bus are processed in the order given by the machine program. Based on monotonicity, we discuss and prove more sophisticated properties of the strictly in-order pipeline, including timing compositionality, which are useful in timing analysis.

Structure of the Thesis and Own Prior Work

In Chapter 2, we formally introduce the basic notions relevant for timing analysis. This includes but is not limited to the notions of abstraction, joining, and compositionality. Our formal definition of timing compositionality has first been published in [Hahn et al., 2015b]. In Chapter 3, we present the overall analysis flow for timing verification. This includes the algorithmic realisation of timing analysis as well as details on an actual implementation. Both chapters present mostly well-known material in a novel way, and thus serve as an introduction to timing analysis which provides the necessary background for this thesis.

We introduce the progress-based abstraction of processor pipelines in Chapter 4. To this end, we extend the formalism described in Chapter 2. In addition, we describe the strictly in-order pipeline in detail and prove its monotonic timing behaviour. We evaluate the performance of the proposed pipeline design. The initial ideas have been developed in [Hahn et al., 2015a]. The formal definition of the strictly in-order pipeline, the relevant proofs, and a thorough evaluation have been published in [Hahn and Reineke, 2018].

In Chapter 5, we first discuss the validity of the timing compositionality assumption. Then, we examine three possible options to achieve compositionality. We evaluate the precision and the efficiency of the presented techniques. The compositional base bound technique has first been discussed and published in [Hahn et al., 2016].

Chapter 6 presents work related to timing analysis. We structure the discussion of related work along the main chapters of this thesis. The individual parts are self-contained and can be read in parallel to the corresponding chapters. In Chapter 7, we conclude and outline future research directions.

Chapter 2

Formal Foundations

Timing analysis is an instance of the more general problem to determine behavioural properties of a given discrete system. The overall approach is to *approximate* the behaviour of the system in an efficient manner and to derive the behavioural properties based on this approximation. In this section, we present the formal concepts used to reason about the soundness of such approximations.

2.1 Setting

We consider systems whose behaviour can be modelled by means of labelled, non-deterministic transition systems as first introduced in [Keller, 1976]. A *configuration*, i.e. a node in this transition system, comprises everything that influences the future behaviour of the given system. We denote the space of configurations by $\mathcal{C}$.

We describe the system behaviour at a discrete time granularity, e.g. at the level of individual processor cycles. A set $\mathcal{E}$ comprises *events* that occur during the transition between two configurations in a single cycle. The system behaviour, given by the relation $cycle \subseteq \mathcal{C} \times 2^{\mathcal{E}} \times \mathcal{C}$, determines the possible transitions between two configurations with the respective set of occurred events. If the system behaves deterministically based on the information in configurations in $\mathcal{C}$, the cycle transition is described by a function $cycle : \mathcal{C} \to 2^{\mathcal{E}} \times \mathcal{C}$. We use the notation $\mathcal{R}(a)(b)(c)$ to express $(a, b, c) \in \mathcal{R}$.

Given a terminating program p, we are interested in the portion of the overall transition system $(\mathcal{C}, cycle)$ that describes the execution of program p. We denote the set of *initial* configurations by $I_p \subseteq \mathcal{C}$ and the set of *final* events by $F_p \subseteq \mathcal{E}$. If a final event is observed during a cycle

transition, we call the successor configuration final as well. We consider those behaviours of the system that start in an initial configuration and end in a final configuration, i.e. after a final event has been observed. We use the directed, labelled *execution graph* $G_p := (V_p, E_p, I_p, F_p)$ to describe the relevant portion of the system behaviour. The nodes $V_p \subseteq \mathcal{C}$ and edges $E_p \subseteq V_p \times 2^{\mathcal{E}} \times V_p$ are the minimal set and relation such that

$$\forall c \in V_p \cup I_p.\ c \text{ not final} \wedge \mathit{cycle}(c)(\mathit{evs})(c') \Leftrightarrow c' \in V_p \wedge E_p(c)(\mathit{evs})(c'). \quad (2.1)$$

The execution graph G_p induces a set of finite traces $\mathcal{T}(G_p)$ through the graph that start in an initial configuration from I_p and end by a final event from F_p:

$$\mathcal{T}(G_p) := \{\tau \in V_p \times (2^{\mathcal{E}} \times V_p)^* \mid \tau_0.c \in I_p \wedge \tau_{|\tau|}.\mathit{evs} \cap F_p \neq \emptyset \wedge \forall\, i \in [1, |\tau|].\ E_p(\tau_{i-1}.c)(\tau_i.\mathit{evs})(\tau_i.c)\}. \quad (2.2)$$

$|\tau|$ denotes the length of the trace, i.e. the number of transitions, or in other words, the number of event-configuration pairs. τ_i denotes the i-th such pair in τ and $\tau_i.c$ ($\tau_i.\mathit{evs}$) the configuration (set of events) of this pair. To simplify notation, $\tau_0.c$ denotes the first configuration in τ and $\tau_0.\mathit{evs} = \emptyset$.

We employ *weight* functions $w : \left(2^{\mathcal{E}}\right)^* \to W$ to characterise individual execution traces based on the occurred events. We require the set of possible weight values W to be a complete lattice with partial order $\leq$ and least upper bound max and greatest lower bound min. As an example, a weight function might calculate the length of a trace or the number of specific events encountered on a trace. In these cases, the set of possible weight values W is $\mathbb{N}_0 \cup \{\infty\}$. For convenience, we write $w(\tau)$ to denote the weight of the events sequence extracted from trace τ. We call a weight *additive* if

$$\forall \tau \in \mathcal{C} \times (2^{\mathcal{E}} \times \mathcal{C})^* \forall i \in [0, |\tau|].\ w(\tau) = w(\tau_{0..i}) + w(\tau_{i..|\tau|}), \quad (2.3)$$

where $\tau_{k..l}$ denotes the subtrace of τ starting at position k with $\tau_k.c$ and ending at position l with τ_l. For the sake of brevity, we focus on the maximisation of additive weights in the following, if not stated otherwise.

2.2 Abstraction

An *explicit* construction of the execution graph G_p and subsequent weight maximisation over traces $\mathcal{T}(G_p)$ is computationally infeasible due to the

system's complexity. To reduce the computational demand, we employ *abstractions*. An abstraction is a compact approximate representation of the system's behaviour as described by G_p. The compactness ensures the computational feasibility of constructing the abstract representation. To achieve compactness, abstraction focuses on the most relevant aspects of the system while leaving other aspects aside. Based on the abstract representation, we compute upper weight bounds.

In this section, we treat the conditions under which an abstraction is *sound*, i.e. correctly approximates the system's behaviour. The ideas that we discuss in the following are found in *abstract interpretation* [Cousot and Cousot, 1977] and *model checking* [Clarke et al., 1994], both of which are frameworks to reason about the soundness of abstractions. In Chapter 3, we provide details on how to actually construct an abstract representation in our specialised setting of timing analysis.

An *abstract configuration* compactly describes a set of concrete configurations. This is formally captured by a *configuration concretisation* function $\gamma^{conf} : \widehat{\mathcal{C}} \to 2^{\mathcal{C}}$ in the sense of abstract interpretation. We denote the set of abstract configurations by $\widehat{\mathcal{C}}$.

There are different possibilities to abstractly represent a set of concrete configurations. In general, the choice of $\widehat{\mathcal{C}}$ impacts the *precision* of the obtainable upper bound and the *efficiency* to construct the abstract representation. We provide examples in Section 3.3 (Microarchitectural Analysis) and Chapter 4 (Progress-based Abstraction).

Besides the abstract configurations, an abstraction includes an *abstract cycle behaviour* operating on these abstract configurations. The abstract cycle behaviour approximates the concrete cycle behaviour $cycle \subseteq \mathcal{C} \times 2^{\mathcal{E}} \times \mathcal{C}$. The configuration abstraction can introduce *uncertainty* to the cycle behaviour. This uncertainty results in non-determinism within the abstract cycle behaviour leading to multiple abstract successor configurations. The abstract cycle behaviour is thus described by a relation $\widehat{cycle} \subseteq \widehat{\mathcal{C}} \times 2^{\mathcal{E}} \times \widehat{\mathcal{C}}$—even if the concrete cycle behaviour is deterministic. We require the abstract cycle relation $\widehat{cycle}$ to be *locally consistent* w.r.t. the concrete cycle behaviour *cycle*, i.e. each concrete cycle transition is captured by an abstract cycle transition:

$$\forall c, c' \in \mathcal{C}\, \forall evs \subseteq \mathcal{E}\, \forall \widehat{c} \in \widehat{\mathcal{C}}.\, c \in \gamma^{conf}(\widehat{c}) \wedge cycle(c)(evs)(c') \\ \Rightarrow \exists \widehat{c}' \in \widehat{\mathcal{C}}.\, \widehat{cycle}(\widehat{c})(evs)(\widehat{c}') \wedge c' \in \gamma^{conf}(\widehat{c}'). \quad (2.4)$$

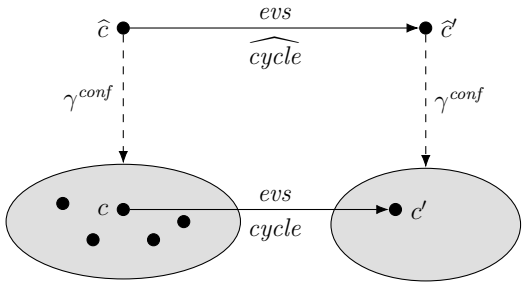


Figure 2.1: Local consistency of the abstract $\widehat{cycle}$ behaviour w.r.t. the concrete behaviour.

Figure 2.1 illustrates the definition of local consistency.

We consider the portion of the overall abstract transition system $(\widehat{\mathcal{C}}, \widehat{cycle})$ that approximates the concrete execution graph $G_p = (V_p, E_p, I_p, F_p)$. We require the initial abstract configurations $\widehat{I}_p \subseteq \widehat{C}$ to cover all concrete initial configurations:

$$\forall i \in I_p \, \exists \widehat{i} \in \widehat{I}_p . \, i \in \gamma^{conf}(\widehat{i}). \tag{2.5}$$

Since we use the same set of events $\mathcal{E}$ in $\widehat{cycle}$ and *cycle*, the final events remain the same, i.e. $\widehat{F}_p = F_p$.

An *abstract execution graph* $\widehat{G}_p := (\widehat{V}_p, \widehat{E}_p, \widehat{I}_p, \widehat{F}_p)$ is a directed, labelled graph that describes the approximate system behaviour which starts from any initial configuration in $\widehat{I}_p$ and ends by observing a final event from $\widehat{F}_p$. The nodes $\widehat{V}_p \subseteq \widehat{\mathcal{C}}$ and edges $\widehat{E}_p$ are the minimal set and relation such that

$$\forall \widehat{c} \in \widehat{V}_p \cup \widehat{I}_p . \, \widehat{c} \text{ not final} \wedge \widehat{cycle}(\widehat{c})(evs)(\widehat{c}') \Leftrightarrow \widehat{c}' \in \widehat{V}_p \wedge \widehat{E}_p(\widehat{c})(evs)(\widehat{c}'). \tag{2.6}$$

An abstract execution graph approximating G_p might not be uniquely defined due to the choice of initial abstract configurations in Equation 2.5.

A graph $\widehat{G}_p$ induces a set of abstract traces denoted by $\mathcal{T}(\widehat{G}_p)$. The concretisation function of traces $\gamma^{traces} : 2^{\widehat{\mathcal{C}} \times \left(2^{\mathcal{E}} \times \widehat{\mathcal{C}}\right)^*} \to 2^{\mathcal{C} \times \left(2^{\mathcal{E}} \times \mathcal{C}\right)^*}$ relates sets of abstract traces to sets of concrete traces. A single abstract trace $\widehat{\tau}$ represents only concrete traces of equal length $|\widehat{\tau}|$. We define the trace

concretisation function by element-wise application of the configuration concretisation γ^{conf}:

$$\gamma^{traces}(\widehat{T}) := \{\tau \mid \exists \widehat{\tau} \in \widehat{T}.\, |\tau| = |\widehat{\tau}| \wedge \forall i \in [0, |\widehat{\tau}|].\, \tau_i.c \in \gamma^{conf}(\widehat{\tau}_i.c) \\ \wedge \tau_i.evs = \widehat{\tau}_i.evs\}. \quad (2.7)$$

We can now prove that the traces through the abstract execution graph approximate all traces through the concrete execution graph.

Theorem 2.2.1 (Trace Coverage)**.** *Let $\widehat{\mathcal{C}}$ be a set of configurations that abstract from $\mathcal{C}$ and $\widehat{cycle}$ a relation which is locally consistent w.r.t. cycle. Let G_p be an execution graph and $\widehat{G}_p$ an abstract execution graph, i.e. satisfying Equations 2.5 and 2.6.*
The set of abstract traces covers *all concrete traces:*

$$\mathcal{T}(G_p) \subseteq \gamma^{traces}(\mathcal{T}(\widehat{G}_p)). \quad (2.8)$$

Proof. The proof can be carried out by structural induction on traces using Equation 2.5 in the base case and Equations 2.4 and 2.6 in the induction step. A proof of the generalised Theorem 2.3.1 is shown in Section 2.3. □

Finally, as a corollary, the maximised weight over abstract traces $\mathcal{T}(\widehat{G}_p)$ constitutes an upper bound on the weights of the concrete traces $\mathcal{T}(G_p)$.

Corollary 2.2.2 (Sound weight bound)**.** *Let the conditions of Theorem 2.2.1 be given. Furthermore, let $w : \left(2^{\mathcal{E}}\right)^* \to W$ be a weight function. The maximum weight w over all traces through the abstract graph provides an* upper bound *on the weights of the concrete traces:*

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) \leq \max_{\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)} w(\widehat{\tau}). \quad (2.9)$$

Proof. Let $\tau \in \mathcal{T}(G_p)$ be the trace with the maximal weight $w(\tau)$. According to Theorem 2.2.1, there is a $\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)$ such that $\tau \in \gamma^{traces}(\{\widehat{\tau}\})$. By definition of γ^{traces}, τ and $\widehat{\tau}$ have the same extracted event sequences and thus $w(\tau) = w(\widehat{\tau})$. Thus, the claim follows by the definition of max. □

Note that due to the employed abstraction, $\mathcal{T}(\widehat{G}_p)$ might contain abstract traces that do not represent any concrete trace. However, such traces might exhibit the maximal weight in the abstract execution graph. We refer to Section 2.4 (Cooperation) for further details.

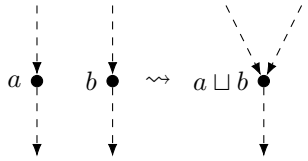


Figure 2.2: Merging of similar configurations reduces the complexity of the abstract execution graph.

2.3 To Join or Not to Join

In the previous section, we introduced *abstraction* in the form of abstract execution graphs that approximate the concrete execution graph. The construction of an abstract execution graph can still be expensive, e.g. due to the non-deterministic splits of the $\widehat{cycle}$ transformer caused by uncertain information in the abstract configurations. However, there is usually redundancy in the graph that we can exploit: separate abstract configurations are similar and exhibit similar subsequent behaviour. It would be beneficial to "merge" such similar configurations into one abstract configuration and thus to examine their behaviour together at once. As an example consider Figure 2.2. Merging complements the aforementioned splitting of configurations in $\widehat{cycle}$ and reduces the complexity of the execution graph, i.e. the number of nodes and edges.

Let $\widehat{\mathcal{C}}$ be the set of abstract configurations. We employ a *partial order* $\sqsubseteq \subseteq \widehat{\mathcal{C}} \times \widehat{\mathcal{C}}$ to describe whether an abstract configuration represents, compared to another one, a subset of the concrete configurations and thus execution behaviours:

$$a \sqsubseteq b \Rightarrow \gamma^{conf}(a) \subseteq \gamma^{conf}(b). \tag{2.10}$$

In other words, the concretisation function $\gamma^{conf} : \widehat{\mathcal{C}} \to 2^{\mathcal{C}}$ is monotonic w.r.t. the partial order $\sqsubseteq$. The partial order thereby models the usual notion of *precision* in the sense of static program analysis [Cousot and Cousot, 1977; Nielson et al., 1999].

In an abstract execution graph $\widehat{G}_p = (\widehat{V}_p, \widehat{E}_p, \widehat{I}_p, \widehat{F}_p)$, we can replace a node $v \in \widehat{V}_p \subseteq \widehat{\mathcal{C}}$ by an abstract configuration w with $v \sqsubseteq w$. As w describes more concrete configurations, a locally consistent abstract cycle behaviour $\widehat{cycle}$ might yield more and different abstract successor configurations for w

than for v. Consequently, the subgraph starting at w differs from the one started at v.

To effectively reduce the graph complexity, multiple different nodes $v_i \in \widehat{V}_p$ should be merged, i.e. replaced by a common configuration w such that $v_i \sqsubseteq w$ for all v_i. We call such a common configuration w an *upper bound* of v_i. If the partially-ordered set of abstract configurations $(\widehat{\mathcal{C}}, \sqsubseteq)$ constitutes a *complete lattice*, each subset of elements has a well-defined *least* upper bound $\bigsqcup : 2^{\widehat{\mathcal{C}}} \to \widehat{\mathcal{C}}$. The least upper bound operator is also called *join* operator. Among all upper bound operators to merge a set of abstract configurations, the join operator provides the most precise result.

An edge in an abstract execution graph $\widehat{G}_p$ now entails a $\widehat{cycle}$ transition followed by an optional configuration replacement according to $\sqsubseteq$. We account for this relaxed edge interpretation by revising Equation 2.6:

$$\forall \widehat{c} \in \widehat{V}_p \cup \widehat{I}_p.\ \widehat{c} \text{ not final} \wedge \widehat{cycle}(\widehat{c})(evs)(\widehat{c}') \Leftrightarrow \exists \widehat{c}'_u \in \widehat{V}_p.\ \widehat{c}' \sqsubseteq \widehat{c}'_u \wedge \widehat{E}_p(\widehat{c})(evs)(\widehat{c}'_u). \quad (2.11)$$

Note that this condition subsumes the previous one as the partial order $\sqsubseteq$ is reflexive. In particular, an abstract execution graph without any merges and replacements remains valid.

Theorem 2.3.1 (Trace Coverage)**.** *Let $\widehat{\mathcal{C}}$ be a set of configurations abstracted from $\mathcal{C}$ and $\widehat{cycle}$ a relation which is locally consistent w.r.t. cycle. Let G_p be an execution graph and $\widehat{G}_p$ an abstract execution graph allowing replacements, i.e. satisfying Equations 2.5 and 2.11. The set of abstract traces* covers *all concrete traces:*

$$\mathcal{T}(G_p) \subseteq \gamma^{traces}(\mathcal{T}(\widehat{G}_p)). \quad (2.12)$$

Proof. Let a concrete trace $\tau \in \mathcal{T}(G_p)$ be given. We need to show that $\tau \in \gamma^{traces}(\mathcal{T}(\widehat{G}_p))$. First, we show that each prefix $\tau^{(i)} = \tau_0 \dots \tau_{i-1}\tau_i$ with $i \leq |\tau|$ has an abstract counterpart $\widehat{\tau}^{(i)}$ in $\widehat{G}_p$ such that $\tau^{(i)} \in \gamma^{traces}(\{\widehat{\tau}^{(i)}\})$. We prove this by induction over i.

Induction base:
We consider the case $i = 0$. By definition, $\tau^{(0)} = \tau_0$ and $\tau_0.c \in I_p$. Equation 2.5 provides an $\widehat{i} \in \widehat{I}_p$ such that $\tau_0.c \in \gamma^{conf}(\widehat{i})$. Choosing $\widehat{\tau}^{(0)} = \widehat{i} \in \widehat{V}_p$ concludes the base case.

Induction step:
Now, we consider the step from i to $i+1$ with $i < |\tau|$. By the *induction hypothesis*, we know that for the concrete prefix $\tau^{(i)}$ there exists an abstract prefix $\widehat{\tau}^{(i)}$ in $\widehat{G}_p$ such that $\tau^{(i)} \in \gamma^{traces}(\{\widehat{\tau}^{(i)}\})$. In particular, we know that $\tau_i.c \in \gamma^{conf}(\widehat{\tau}_i.c)$. By definition and $i < |\tau|$, we also know that $cycle(\tau_i.c)(\tau_{i+1}.evs)(\tau_{i+1}.c)$ and $\tau_i.c$ is not final.

Using local consistency of $\widehat{cycle}$, we get a configuration $\widehat{c}' \in \widehat{\mathcal{C}}$ such that $\widehat{cycle}(\widehat{\tau}_i.c)(\widehat{\tau}_{i+1}.evs)(\widehat{c}')$ with $\tau_{i+1}.evs = \widehat{\tau}_{i+1}.evs$ and $\tau_{i+1}.c \in \gamma^{conf}(\widehat{c}')$. Applying the $\Rightarrow$ part of Equation 2.11, we obtain a configuration $\widehat{c}'_u \in \widehat{V}_p$ with $\widehat{E}_p(\widehat{\tau}_i.c)(\widehat{\tau}_i.evs)(\widehat{c}'_u)$. Furthermore, we obtain $\widehat{c}' \sqsubseteq \widehat{c}'_u$ and by Equation 2.10, $\tau_{i+1}.c \in \gamma^{conf}(\widehat{c}'_u)$. Using $\widehat{\tau}_{i+1}.c = \widehat{c}'_u$ concludes the induction step.

Finally, it remains to show that the candidate trace $\widehat{\tau} = \widehat{\tau}^{(|\tau|)}$ is an actual member of $\mathcal{T}(\widehat{G}_p)$, i.e. $\widehat{\tau}$ ends with a final event. As $\tau \in \mathcal{T}(G_p)$ and $\tau \in \gamma^{traces}(\{\widehat{\tau}\})$, we get $\widehat{\tau}_{|\widehat{\tau}|}.evs \cap \widehat{F}_p \neq \emptyset$. This concludes the overall proof. □

Different replacements of nodes lead to different, sound abstract execution graphs $\widehat{G}_p$. Thus, $\widehat{G}_p$ is not uniquely defined by the choice of initial states $\widehat{I}_p$ any more. The choice which configurations to join trades off the *precision* of the resulting graph against the *efficiency* of the graph construction. We provide details on a possible join strategy in Section 3.3 (Low-Level Analysis).

2.4 Cooperation

In general, we employ several different abstractions. Each abstraction focuses on a specific aspect of the concrete system behaviour, e.g. the values computed during execution or the evolution of the microarchitectural state. To obtain precise results, the different abstractions cooperate by exchanging information.

Consider n abstractions with the respective abstract execution graphs $\widehat{G}_1, \ldots, \widehat{G}_n$ such that the traces of each $\widehat{G}_i$ approximate the concrete traces through G_p via γ_i^{traces}. If all abstractions satisfy the trace coverage property

in Equation 2.8, the combined graph $\widehat{G} = \prod_{i=1}^{n} \widehat{G}_i$ with the following concretisation function satisfies it as well

$$\gamma^{traces}\left(\mathcal{T}(\widehat{G})\right) = \gamma^{traces}\left(\mathcal{T}((\widehat{G}_1, \ldots, \widehat{G}_n))\right) = \bigcap_{i=1}^{n} \gamma_i^{traces}(\mathcal{T}(\widehat{G}_i)). \quad (2.13)$$

Note that if some abstract execution graph $\widehat{G}_i$ does not fulfil Equation 2.8, it is unsound to take the intersection as it might miss the trace leading to the maximal weight.

In practice, it is computationally infeasible to first concretise and then compute the intersection. Instead, we want to perform the intersection efficiently on the abstract execution graphs directly.

Infeasible Abstract Traces We call an abstract trace $\widehat{\tau} \in \mathcal{T}(\widehat{G}_j)$ *infeasible* if

$$\gamma^{traces}(\{\widehat{\tau}\}) \cap \mathcal{T}(G_p) = \emptyset. \quad (2.14)$$

Infeasible abstract traces arise due to the approximative character of abstractions.

A sufficient criterion for infeasibility based on cooperation with other abstractions is

$$\gamma^{traces}(\{\widehat{\tau}\}) \cap \gamma_i^{traces}(\mathcal{T}(\widehat{G}_i)) = \emptyset, \quad (2.15)$$

as $\gamma_i^{traces}(\mathcal{T}(\widehat{G}_i)) \supseteq \mathcal{T}(G_p)$. For soundness in the sense of Equation 2.8, it is sufficient to only consider feasible traces.

In practice, there are two possibilities to prune abstract infeasible traces: Either when constructing $\widehat{G}_j$ or when searching the maximal-weight trace through $\widehat{G}_j$. In both cases, we use the (number of) occurrences of specific events to exchange information between abstractions. In the first case, we can prune a successor configuration of $\widehat{cycle}$ if this cycle transition emits an event that does not occur in $\widehat{G}_i$. In the second case, we formulate constraints for the search of the maximal-weight trace. As an example, we exclude traces through $\widehat{G}_j$ that exhibit more events of a certain type than any trace through $\widehat{G}_i$.

We provide concrete examples and the realisation of such constraints in Section 3.3 (Low-Level Analysis) explaining the implementation aspects. More information about the formal background on pruning infeasible abstract traces as well as its applications in the analysis of multi-core systems can be found in [Jacobs, 2014, 2018; Jacobs et al., 2016].

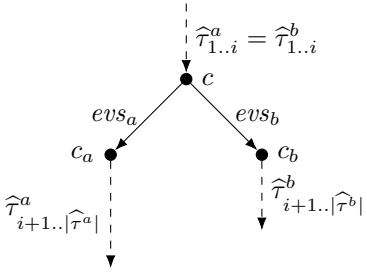


Figure 2.3: Excerpt from an abstract execution graph illustrating the notion of domination.

2.5 Domination

Up to now, we have studied techniques to approximate the set of concrete execution traces which guarantee trace coverage (Equation 2.8) and consequently sound weight bounds (Equation 2.9). The idea behind domination is that it is not necessary to construct a full abstract execution graph that covers *all* concrete traces. Rather, it is sufficient to construct the part of an abstract graph that contains a maximal-weight abstract trace. As a consequence, such a *partial* abstract execution graph does not satisfy the trace coverage, but the sound weight bound property.

Intuitively, upon a non-deterministic choice during a cycle transition, there is often one seemingly *worst* configuration among the successor configurations that should lead to a maximal-weight trace. If we can prove that this intuition is right, we can reduce the complexity of the abstract execution graph by only considering such worst successor configurations.

First, we need to formalise our notion of a successor configuration c_a being "worse" than c_b. As an example, consider the fragment of an abstract execution graph in Figure 2.3. We say configuration c_a *dominates* c_b—alternatively cycle transition $\widehat{E}(c)(evs_a)(c_a)$ dominates $\widehat{E}(c)(evs_b)(c_b)$—if each abstract trace $\widehat{\tau}^b$ through $\widehat{E}(c)(evs_b)(c_b)$ is outweighed by a trace $\widehat{\tau}^a$ through $\widehat{E}(c)(evs_a)(c_a)$ w.r.t. a given weight w.

Definition 2.5.1 (Configuration Domination)**.** Let a weight function $w : \left(2^{\mathcal{E}}\right)^* \to W$ and an abstract execution graph $\widehat{G}_p = (\widehat{V}_p \subseteq \widehat{\mathcal{C}}, \widehat{E}_p, \widehat{I}_p, \widehat{F}_p)$ be given. Furthermore, let $c, c_a, c_b \in \widehat{V}_p$ be configurations such that $\widehat{E}_p(c)(evs_a)(c_a)$ and $\widehat{E}_p(c)(evs_b)(c_b)$.
We say c_a *dominates* c_b w.r.t. w if and only if

$$\begin{aligned}\forall \widehat{\tau}^b \in \mathcal{T}(\widehat{G}_p)\ \forall i.\ \widehat{\tau}^b_i.c = c \wedge \widehat{\tau}^b_{i+1} = (evs_b, c_b) \Rightarrow \\ \exists \widehat{\tau}^a \in \mathcal{T}(\widehat{G}_p).\ \widehat{\tau}^a_{1..i} = \widehat{\tau}^b_{1..i} \wedge \widehat{\tau}^a_{i+1} = (evs_a, c_a) \wedge \\ \forall \tau^b \in \gamma^{traces}(\{\widehat{\tau}^b\}).\ w(\tau^b) \leq w(\widehat{\tau}^a). \quad (2.16)\end{aligned}$$

In a scenario as described in Definition 2.5.1, an abstract trace containing the edge $\widehat{E}_p(c)(evs_b)(c_b)$ can never *solely* lead to the maximal weight. Thus, it is sound to only follow the edge to successor c_a and to ignore the edge to successor c_b. The subgraph that starts in c_b and is not reachable by any other initial configuration can be pruned from the abstract execution graph.

For technical reasons, we assume that no configuration joining (Section 2.3) has been performed in the subgraph that starts in c_b. Joining a node of this subgraph with another configuration could introduce new, potentially infeasible traces through c_b such that their weight exceeds the weight of any $\widehat{\tau}^a$. Thus, joining would void the domination property. Note however, that joining can still be employed in the remaining execution graph. A practical algorithm to construct the abstract execution graph, e.g. as described in Section 3.3 (Low-Level Analysis), never creates the subgraph starting in c_b anyway.

Theorem 2.5.2 (Sound Weight, Domination)**.** *Let $\widehat{\mathcal{C}}$ be a set of configurations abstracted from $\mathcal{C}$ and $\widehat{cycle}$ a relation locally consistent to cycle. Let c_a and c_b be abstract successor configurations of c such that c_a dominates c_b. Let G_p be an execution graph and $\widehat{G}_p$ a partial abstract execution graph, i.e. $\widehat{G}_p$ satisfies Equations 2.5 and 2.6 for $\widehat{c}' \neq c_b$. Furthermore, let $w : \left(2^{\mathcal{E}}\right)^* \to W$ be a weight function. The maximum of w on traces through the abstract graph provides an* upper bound *on weights on concrete traces:*

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) \leq \max_{\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)} w(\widehat{\tau}). \quad (2.17)$$

Proof. Let $\tau \in \mathcal{T}(G_p)$ be a concrete trace with maximal weight. In a full abstract execution graph, there is a corresponding abstract trace $\widehat{\tau}$ such that $\tau \in \gamma^{traces}(\{\widehat{\tau}\})$ and $w(\tau) \leq w(\widehat{\tau})$. If $\widehat{\tau}$ does not contain a transition from c to c_b, then $\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)$ and the claim follows. If $\widehat{\tau}$ does contain the transition from c to c_b, then it is not a trace through the partial graph $\widehat{G}_p$. However, due to domination there is a trace $\widehat{\tau}^a$ such that $w(\tau) \leq w(\widehat{\tau}^a)$. As $\widehat{\tau}^a$ does not contain the transition from c to c_b, it is a trace through the partial graph $\widehat{G}_p$. □

Note that the domination property is always linked to a given weight function. This is important, because the fact that c_a dominates c_b w.r.t. w_1 does not imply domination w.r.t. w_2. As an example, consider w_1 to assess the execution time while w_2 assesses the number of cache hits. Thus, in a setting where multiple weights are maximised and/or minimised, the pruning described above might not be applicable.

Proving domination for individual situations $c, c_a, c_b, evs_a, evs_b$ is tedious and the savings during graph construction are small. Instead, we want to prove domination once for many similar situations, e.g. all splits caused by a specific kind of non-determinism. To reason about specific kinds of non-determinism, we use *labels* to group similar events that describe similar causes of a split. Formally, a set of labels $L \subseteq 2^{\mathcal{E}}$ partitions a subset of all events:

$$\forall l, l' \in L.\ l \cap l' \neq \emptyset \Rightarrow l = l'. \tag{2.18}$$

Definition 2.5.3 (Label/Weight Domination)**.** Let $\{l_a, l_b, \ldots\} \subseteq 2^{\mathcal{E}}$ be a set of labels that describe a non-deterministic choice (Equation 2.18). Furthermore, let $w : \left(2^{\mathcal{E}}\right)^* \to W$ be a weight function. We say l_a *dominates* l_b w.r.t. weight w if and only if for all abstract execution graphs $\widehat{G}_p$

$$\begin{aligned} &\forall c, c_b \in \widehat{V}_p.\ \widehat{E}(c)(evs_b)(c_b) \wedge l_b \cap evs_b \neq \emptyset \Rightarrow \\ &\qquad \exists c_a \in \widehat{V}_p.\ \widehat{E}(c)(evs_a)(c_a) \wedge l_a \cap evs_a \neq \emptyset \wedge c_a \text{ dominates } c_b \text{ w.r.t. } w. \end{aligned} \tag{2.19}$$

Alternatively, we say w^* dominates w, where w^* is a partially defined weight function that coincides with w on traces without l_b and is undefined otherwise.

Based on this extended definition, an algorithm to construct the abstract execution graph can decide *locally* based on l_b to ignore the associated successor configurations.

Upon a non-deterministic choice, one label $l_a \in \{l_a, l_b, \ldots\}$ is often intuitively considered as the "locally worst case", i.e. l_a exhibits the largest weight in a locally bounded part of the following execution. According to Definition 2.5.3, a single scenario $c, c_b \in \widehat{\mathcal{C}}$ such that no $c_a \in \widehat{\mathcal{C}}$ (globally) dominates c_b causes the locally worst case l_a to *not* dominate l_b. Such a counter-intuitive scenario is commonly referred to as *timing anomaly*. In other words, a timing anomaly describes a situation in which the "locally worst" decision does not imply the globally worst case. We call a situation *domino effect* [Lundqvist and Stenström, 1999], if the difference in weight among the possible cases $\{l_a, l_b, \ldots\}$ is not bounded by a constant.

Abstractions of complex microarchitectures are known to be prone to timing anomalies such as scheduling anomalies or speculation anomalies [Lundqvist and Stenström, 1999; Reineke et al., 2006]. Note that the definitions of domination and timing anomaly inherently depend on the chosen abstraction $\widehat{\mathcal{C}}$. Abstractions with *and* without anomalous behaviour can exist for the *same* microarchitecture.

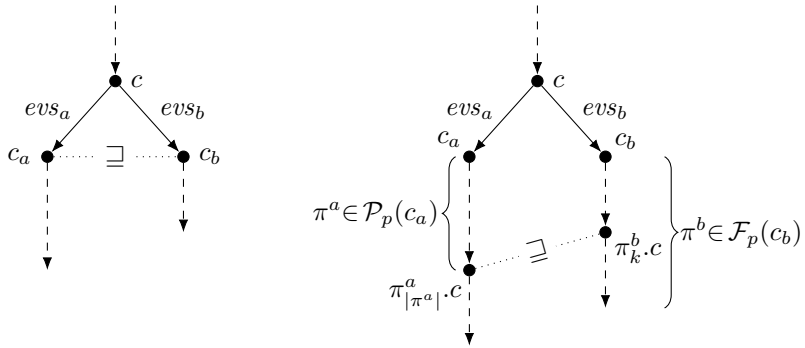
Finally, we present a sufficient condition for domination that can be checked by examination of the cycle transition relation. Consider Figure 2.4a for an example. The intuition behind the sufficient condition is that the traces starting from the locally worst successor configuration c_a cover all traces starting from any other successor c_b.

Theorem 2.5.4. *Let the set of abstract configurations $\widehat{\mathcal{C}}$, the abstract cycle behaviour $\widehat{cycle} \subseteq \widehat{\mathcal{C}} \times 2^{\mathcal{E}} \times \widehat{\mathcal{C}}$, an additive weight function $w : \left(2^{\mathcal{E}}\right)^* \to W$, and $\{l_a, l_b, \ldots\} \subseteq 2^{\mathcal{E}}$ a set of labels describing a non-deterministic choice in $\widehat{cycle}$ (Equation 2.18) be given. Furthermore, let $\sqsubseteq \subseteq \widehat{\mathcal{C}} \times \widehat{\mathcal{C}}$ be a partial order consistent with γ^{conf} (Equation 2.10).*
l_a dominates *l_b w.r.t. weight w if*

$$\begin{aligned}\forall c, c_b \in \widehat{\mathcal{C}}.\ \widehat{cycle}(c)(evs_b)(c_b) \wedge l_b \cap evs_b \neq \emptyset \Rightarrow \\ \exists c_a \in \widehat{\mathcal{C}}.\ \widehat{cycle}(c)(evs_a)(c_a) \wedge l_a \cap evs_a \neq \emptyset \wedge \\ c_b \sqsubseteq c_a \wedge w(c \circ (evs_b, c_b)) \leq w(c \circ (evs_a, c_a)).\end{aligned} \tag{2.20}$$

Proof. See generalised Theorem 2.5.5. □

As a consequence of Theorem 2.5.4, an abstraction that satisfies the above condition is *free of anomalies* w.r.t. l_a and l_b and the given weight w. The condition can thus be used to formally verify the absence of timing



(a) Theorem 2.5.4: Matching immediate successor configurations.

(b) Relaxed Theorem 2.5.5: Matching subsequent configurations.

Figure 2.4: Excerpt from an abstract execution graph illustrating the sufficient condition for domination.

anomalies of a given microarchitectural abstraction. Unfortunately, it is generally hard to find a microarchitecture such that an abstraction is free of anomalies. In Chapter 4 (Progress-based Abstraction), we propose a hardware design and an abstraction thereof that satisfies the condition in Theorem 2.5.4.

The condition in Theorem 2.5.4 is restrictive as the *immediate* successor configurations are matched. It is however sufficient to match some configuration on *each* partial final trace starting from c_b with some configuration on *some* partial trace starting from c_a. Figure 2.4b illustrates this generalisation. The set of partial traces starting from a configuration c is defined as

$$\mathcal{P}(c) := \{\widehat{\tau} \in \widehat{\mathcal{C}} \times (2^{\mathcal{E}} \times \widehat{\mathcal{C}})^* \mid \widehat{\tau}_0.c = c \wedge \forall\, i \in [1, |\widehat{\tau}|].\ \widehat{cycle}(\widehat{\tau}_{i-1}.c)(\widehat{\tau}_i.evs)(\widehat{\tau}_i.c)\}. \quad (2.21)$$

For a given program p, we restrict the partial traces to the actual execution of p:

$$\mathcal{P}_p(c) := \{\widehat{\tau} \in \mathcal{P}(c) \mid \forall\, i \in [1, |\widehat{\tau}| - 1].\ \widehat{\tau}_i.evs \cap \widehat{F}_p = \emptyset\}. \quad (2.22)$$

We define the set of partial *final* traces, i.e. the partial traces that end with a final event $\widehat{F}_p$, as

$$\mathcal{F}_p(c) := \{\widehat{\tau} \in \mathcal{P}_p(c) \mid \widehat{\tau}_{|\widehat{\tau}|}.evs \cap \widehat{F}_p \neq \emptyset\}. \tag{2.23}$$

We use this relaxed condition when proving compositionality properties in Chapter 5 (Achieving Timing Compositionality).

Theorem 2.5.5. *Let the set of abstract configurations $\widehat{\mathcal{C}}$, the abstract cycle behaviour $\widehat{cycle} \subseteq \widehat{\mathcal{C}} \times 2^{\mathcal{E}} \times \widehat{\mathcal{C}}$, an additive weight function $w : \left(2^{\mathcal{E}}\right)^* \to W$, and $\{l_a, l_b, \ldots\} \subseteq 2^{\mathcal{E}}$ a set of labels describing a non-deterministic choice in $\widehat{cycle}$ (Equation 2.18) be given. Furthermore, let $\sqsubseteq \subseteq \widehat{\mathcal{C}} \times \widehat{\mathcal{C}}$ be a partial order consistent with γ^{conf} (Equation 2.10).*
l_a dominates *l_b w.r.t. weight w if for all programs p*

$$\begin{aligned}
&\forall c, c_b \in \widehat{\mathcal{C}}.\ \widehat{cycle}(c)(evs_b)(c_b) \wedge l_b \cap evs_b \neq \emptyset \Rightarrow \\
&\quad \exists c_a \in \widehat{\mathcal{C}}.\ \widehat{cycle}(c)(evs_a)(c_a) \wedge l_a \cap evs_a \neq \emptyset \wedge \\
&\qquad \forall \pi^b \in \mathcal{F}_p(c_b)\ \exists k \geq 0\ \exists \pi^a \in \mathcal{P}_p(c_a).\ \pi^b_k.c \sqsubseteq \pi^a_{|\pi^a|}.c \wedge \\
&\qquad\quad w\left(c \circ (evs_b, c_b) \circ \pi^b_{0..k}\right) \leq w\left(c \circ (evs_a, c_a) \circ \pi^a\right).
\end{aligned} \tag{2.24}$$

Proof. We need to prove that l_a dominates l_b given that the above criterion is fulfilled. Let an abstract execution graph $\widehat{G}_p$ with configurations c and c_b be given such that $\widehat{cycle}(c)(evs_b)(c_b)$ and $l_b \cap evs_b \neq \emptyset$. Our condition above provides a witness configuration c_a such that $\widehat{cycle}(c)(evs_a)(c_a)$ and $l_a \cap evs_a \neq \emptyset$. For a graphical representation of the described scenario, consider Figure 2.4b.

Using the witness c_a in the definition of label domination, it remains to be shown that c_a dominates c_b. Thus, let $\widehat{\tau}^b \in \mathcal{T}(\widehat{G}_p)$ be a trace through $c = \widehat{\tau}^b_i.c$ and $c_b = \widehat{\tau}^b_{i+1}.c$. We need to show the existence of a trace $\widehat{\tau}^a \in \mathcal{T}(\widehat{G}_p)$ such that

$$\widehat{\tau}^a_{1..i} = \widehat{\tau}^b_{1..i} \wedge \widehat{\tau}^a_{i+1} = (evs_a, c_a) \wedge \forall \tau^b \in \gamma^{traces}(\{\widehat{\tau}^b\}).w(\tau^b) \leq w(\widehat{\tau}^a).$$

Using the remaining assumption of the theorem with $\pi^b = \widehat{\tau}^b_{i+1..|\widehat{\tau}^b|}$, we obtain a $k \geq 0$ and a partial trace π^a such that

$$\widehat{\tau}^b_{i+1+k}.c \sqsubseteq \pi^a_{|\pi^a|}.c.$$

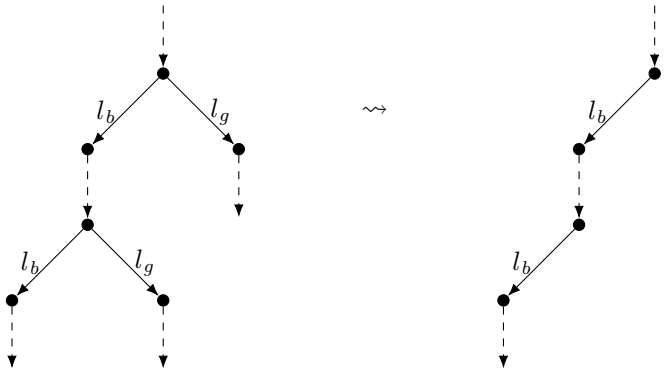


Figure 2.5: Excerpt from an abstract execution graph. On the right after pruning dominated configurations. Excluding the infeasible trace with multiple events from l_b yields an empty set of traces.

Joining both configurations would result in $\pi^a_{|\pi^a|}.c$. By the correctness of joining, we obtain a trace $\tau^{\sharp,b}$ that describes at least the same concrete traces as $\widehat{\tau}^b$ and thus $w(\tau^b) \leq w(\tau^{\sharp,b})$ for any $\tau^b \in \gamma^{traces}(\{\widehat{\tau}^b\})$.

Now, we can construct an abstract trace $\widehat{\tau}^a$ by replacing the part $c \circ (evs_b, c_b) \circ \pi^b_{0..k}$ of $\tau^{\sharp,b}$ by $c \circ (evs_a, c_a) \circ \pi^a$. By using additivity of w and the theorem assumption

$$w\left(c \circ (evs_b, c_b) \circ \pi^b_{0..k}\right) \leq w\left(c \circ (evs_a, c_a) \circ \pi^a\right),$$

we conclude for all $\tau^b \in \gamma^{traces}(\{\widehat{\tau}^b\})$

$$w(\tau^b) \leq w(\tau^{\sharp,b}) \leq w(\widehat{\tau}^a). \qquad \square$$

Domination and Cooperation Cooperation of abstractions requires that *each* abstraction satisfies the trace-coverage condition in Equation 2.8. If one of them only satisfies weight correctness in Equation 2.9, cooperation might yield an unsound result. Thus, domination and cooperation cannot be combined in general.

Assume an abstract trace $\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)$ provides the maximal weight such that Equation 2.9 holds. Furthermore, assume that other abstract traces that provide a sufficiently high weight have been pruned from $\widehat{G}_p$ due

to *domination*. In this case, the trace-coverage condition is not fulfilled for $\mathcal{T}(\widehat{G}_p)$. If a cooperating abstraction detects trace $\widehat{\tau}$ to be infeasible, the resulting set of traces $\mathcal{T}(\widehat{G}_p) \setminus \{\widehat{\tau}\}$ does not fulfil Equation 2.9 any more, thus yielding unsound results.

As an example, consider Figure 2.5 and assume that a cooperating abstraction learns that at most one event of label l_b can occur. Consequently, the worst *feasible* trace has to encounter label l_g at least once. However, if l_b dominates l_g, the l_g successor configurations have been ignored during the graph construction which results in the graph on the right of Figure 2.5. By excluding infeasible traces via cooperation, the trace exhibiting two events of l_b—which is essential to overall correctness—would be pruned.

2.6 Compositionality

In the previous sections, we discussed techniques that tackle the problem of efficiently computing the maximal (minimal) weight of any trace through an (abstract) execution graph. This section deals with compositionality, i.e. the ability to decompose the problem into smaller sub-problems whose partial solutions can be used to derive an overall solution. The sub-problems can hopefully be solved more efficiently than the original problem.

Let a set C of configurations, either concrete or abstract, and the associated cycle behaviour $cycle \subseteq \mathcal{C} \times 2^{\mathcal{E}} \times \mathcal{C}$ be given. $\mathcal{T}(G_p)$ denotes the set of traces through any (abstract) execution graph G_p of any program. Furthermore, we consider a weight $w : \left(2^{\mathcal{E}}\right)^* \to W$ that we want to maximise.

A system under analysis has different constituents that contribute their share to the weight $w(\tau)$ of any trace $\tau \in \mathcal{T}(G_p)$. The contribution of constituent i is given by $wc_i : \left(2^{\mathcal{E}}\right)^* \to W_i$, where W_i and W do not necessarily coincide. The individual contributions are combined by a monotonic composition operator $\bigoplus : \prod_{i=1}^{n} W_i \to W$.

Definition 2.6.1 (Decomposition)**.** Let a weight function $w : \left(2^{\mathcal{E}}\right)^* \to W$ be given. A family of contribution functions $(wc_i : \left(2^{\mathcal{E}}\right)^* \to W_i)_{i=1..n}$ together with a monotonic composition operator $\bigoplus : \prod_{i=1}^{n} W_i \to W$ forms a *decomposition* of weight w if for any execution graph G_p

$$\forall \tau \in \mathcal{T}(G_p).\, w(\tau) \leq \bigoplus_{i=1}^{n} wc_i(\tau). \tag{2.25}$$

If Inequality 2.25 satisfies equality, i.e. the composed contributions equal the weight w for each trace, the decomposition is referred to as *fully compositional*.

In the end, each contribution wc_i is approximated by an individual analysis $Analysis_i$ such that by monotonicity of $\bigoplus$:

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) \leq \max_{\tau \in \mathcal{T}(G_p)} \bigoplus_{i=1}^{n} wc_i(\tau) \tag{2.26}$$

$$\leq \bigoplus_{i=1}^{n} \max_{\tau \in \mathcal{T}(G_p)} wc_i(\tau) \tag{2.27}$$

$$\leq \bigoplus_{i=1}^{n} Analysis_i(p). \tag{2.28}$$

The first inequality 2.26 holds by definition and models the inherent pessimism introduced by the decomposition. The second inequality 2.27 holds by monotonicity of $\bigoplus$. It models the pessimism introduced by the separate maximisation of the weight contributions: If the traces with the respective worst wc_i do not coincide, no trace $\tau \in \mathcal{T}(G_p)$ can maximise *all* weight contributions. The third inequality 2.28 holds by monotonicity of $\bigoplus$ and the soundness of the individual abstractions. It models the pessimism introduced by the respective abstraction used in $Analysis_i$.

Definition 2.6.1 introduces the decomposition of a weight w at the level of a single execution trace. During the analysis, we are finally interested in the maximal weight over *all* execution traces. Thus, it suffices if the maximal weight is bounded by the combination of the respective maximal weight contributions. We call this relaxed variant max-*decomposition*.

Definition 2.6.2 (max-Decomposition)**.** Let $w : \left(2^{\mathcal{E}}\right)^* \to W$ be a weight function. A family of contribution functions $(wc_i : \left(2^{\mathcal{E}}\right)^* \to W_i)_{i=1..n}$ together with a monotonic composition operator $\bigoplus : \prod_{i=1}^{n} W_i \to W$ forms a max-*decomposition* of weight w if for any execution graph G_p

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) \leq \bigoplus_{i=1}^{n} \max_{\tau \in \mathcal{T}(G_p)} wc_i(\tau). \tag{2.29}$$

Note that a decomposition at the level of single execution traces also constitutes a max-decomposition. Throughout this thesis, we usually do

not distinguish both variants and thus we use the term decomposition to refer to Definition 2.6.2.

An interesting instance of compositionality is *timing* compositionality, i.e. (discrete) time as weight. The first formal definition of compositionality in the context of execution time analysis has been introduced in [Hahn et al., 2015b]. In later sections, we provide details on the use of timing compositionality including example decompositions in Section 3.4 (Scheduling Interface and Compositionality) as well as a discussion how we can achieve and prove compositionality in Chapter 5 (Achieving Timing Compositionality).

2.7 Composability

Despite the shared etymological roots, compositionality and composability denote two different concepts in timing analysis. In order to discriminate both terms, we give a definition of composability in this section.

The computation of a weight $w : \left(2^{\mathcal{E}}\right)^* \to W$ on an execution trace does not necessarily require all aspects captured by full configurations $\mathcal{C}$ along the traces. In case the computation is or can be made independent of parts of the configurations, we call the weight w *composable* w.r.t. these parts.

Formally, composability is a type of abstraction in the sense of Section 2.2 that *projects* only a part of the configurations.

Definition 2.7.1 (Composability)**.** Let $\mathcal{C} = \mathcal{C}_r \times \mathcal{C}_i$ be a set of configurations and $w : \left(2^{\mathcal{E}}\right)^* \to W$ a weight function. We call w *composable* w.r.t. $\mathcal{C}_i$ if the following holds: For all execution graphs $G_p = (V_p \subseteq \mathcal{C}, E_p, I_p, F_p)$ of any program, and abstract execution graph $\widehat{G}_p = (\widehat{V}_p \subseteq \mathcal{C}_r, \widehat{E}_p, \widehat{I}_p, \widehat{F}_p)$ with $\gamma^{conf}(c_r) = \{(c_r, c_i) \mid c_i \in \mathcal{C}_i\}$

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) \leq \max_{\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)} w(\widehat{\tau}). \tag{2.30}$$

Stronger notions of composability exist, e.g. that require the result not only to be sound, but to be *tight*:

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) = \max_{\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)} w(\widehat{\tau}). \tag{2.31}$$

An even stronger notion requires the cycle behaviour to be inherently independent of $\mathcal{C}_i$:

$$\exists c_i' \in \mathcal{C}_i \, \exists evs_i \subseteq \mathcal{E}.\ cycle((c_r, c_i))(evs_r \cup evs_i)((c_r', c_i')) \\ \Leftrightarrow \widehat{cycle}(c_r)(evs_r)(c_r'). \quad (2.32)$$

In contrast to these stronger notions, Definition 2.7.1 allows a weight w to be composable w.r.t. $\mathcal{C}_i$ while the concrete system has originally not been designed for composability, i.e. the concrete behaviour *cycle* actually depends on $\mathcal{C}_i$. As an example, composability can be achieved by an abstract behaviour $\widehat{cycle}$ that uses non-deterministic splits to conservatively approximate the possible influence of $\mathcal{C}_i$ on the system's behaviour.

Chapter 3

Timing Analysis

In this chapter, we instantiate the formal foundations of Chapter 2 for the timing analysis problem. We provide an example system and give details on the respective configurations and events. Furthermore, we explain how we perform the actual timing analysis. This includes the actual *construction* of an abstract execution graph to model the behaviour of a program.

3.1 System under Analysis

A system consists of a set of software *tasks* that execute on a hardware platform and interact with the physical environment via sensors and actuators. In addition to the actual machine program, a task comprises parameters that specify its (timing) requirements. In Figure 3.1, we depict a generic modern hardware platform.

The software tasks are initially placed in the common memory. The cores of the processor execute those tasks. A core loads the necessary code and

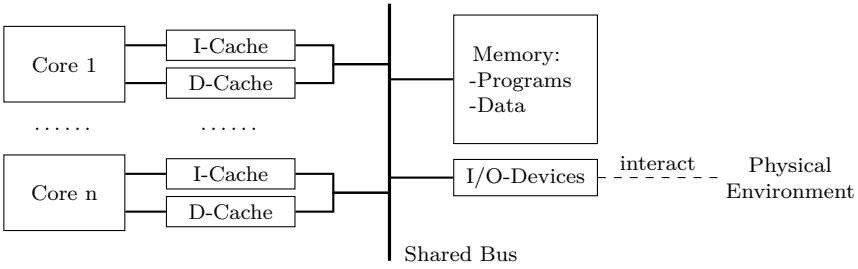


Figure 3.1: Overview of the components of a generic hardware platform.

data from the memory via a shared bus. Caches for instructions and data are employed to bridge the performance gap between the fast cores and the slow memory. I/O-devices allow the cores to interact with the physical environment.

One of the tasks is the *operating system*. The operating system task is the first task that is executed after power-up. It schedules the execution of all other tasks. A periodic timer invokes the operating system task on each core via interrupt to allow for non-cooperative scheduling decisions. For the sake of simplicity, we treat the operating system as an ordinary task.

The space of concrete *configurations* of our generic example system in Figure 3.1 can be described on a high-level as

$$\mathcal{C} := (\mathit{Core} \times \mathit{Caches})^n \times \mathit{SharedBus} \times \mathit{Memory} \times \mathit{PhysicalEnvironment}.$$

A *Core* features for example a set of registers with recent values used for execution, the state of its pipeline, and the state of the timer. One register, the *program counter*, points to the next machine instruction to be executed. A *Cache* comprises the data buffered from memory as well as its management structures such as the state of the replacement policy. The *SharedBus* describes the state of the arbiter that arbitrates the access requests to the memory. The *Memory* includes the values placed in memory as well as the state of the controller that serves accesses. The *PhysicalEnvironment* comprises the state of the physical world which is sensed and modified through the I/O devices.

A configuration $c \in \mathcal{C}$ encompasses everything that influences the behaviour of the above system. The behaviour at the granularity of processor cycles is given by a relation $\mathit{cycle} \subseteq \mathcal{C} \times 2^{\mathcal{E}} \times \mathcal{C}$. Example events are the release and completion of a task, a cache hit or miss of an access, or the blocking of an access at the shared bus. The precise definition of $\mathcal{C}$, $\mathcal{E}$, and *cycle* depends on the actual system under analysis. Providing these definitions for an entire system is out of the scope of this thesis. In Appendix A (Computer Architecture: Concepts), we describe the behaviour of modern hardware components, including a formal definition of the behaviour of an in-order pipeline. More details on the precise modelling of actual complex systems, e.g. based on the PowerPC 755, can be found in the dissertation of Thesing [2004]. For a broader overview of hardware systems, we refer to [Hennessy and Patterson, 2012].

During the analysis, we will focus on the behaviour of single tasks. We call a configuration *initial* for task t if t has just been released for execution

by the operating system task. We call a configuration *final* for task t if its last instruction has left the core and all effects of t have manifested themselves in memory. Based on these notions of initial and final, the concrete execution graph G_t is determined.

3.2 Analysis Overview

In timing verification, we want to prove the timeliness of a given system. Each task of the system has a deadline, which is dictated by the physical environment. *Timing analysis* calculates—per task—an upper bound on the so-called response times, i.e. the maximal time between the release of the task and the completion of the task. If the computed bounds do not exceed the respective deadlines, the timeliness of the system is guaranteed.

Formally, we define the *worst-case response time* of a task t as the maximal length of any trace through the concrete execution graph G_t:

$$WCRT_t := \max_{\tau \in \mathcal{T}(G_t)} w_{time}(\tau) = \max_{\tau \in \mathcal{T}(G_t)} |\tau|.$$

Multiple factors influence the response time of a task t, e.g.

- the inputs provided by previous tasks or sensors observing the physical environment,
- the microarchitectural state including cache contents and pipeline occupancy, and
- the interference from co-running tasks that compete for the access to the shared resources.

Which tasks run concurrently on other cores or which tasks preempt the task t depends on the *scheduling policy* used inside the operating system task.

The construction of an execution graph that takes the scheduling policy and the resulting interleaving of tasks into account is commonly considered computationally infeasible. The de-facto standard approach to timing analysis is to *decompose* the problem in the sense of Section 2.6 (Compositionality). First, we analyse the behaviour of a single task in isolation. Second, we check the schedulability of all tasks with the given scheduling policy while taking the interference effects of co-running tasks into

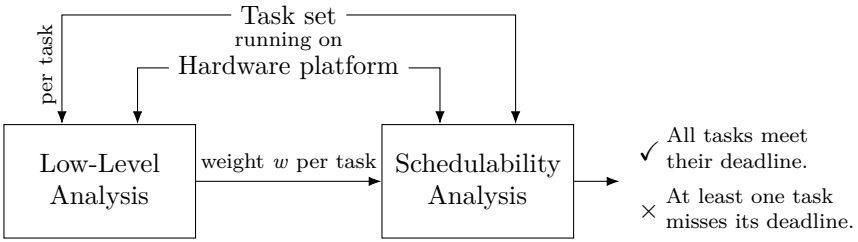


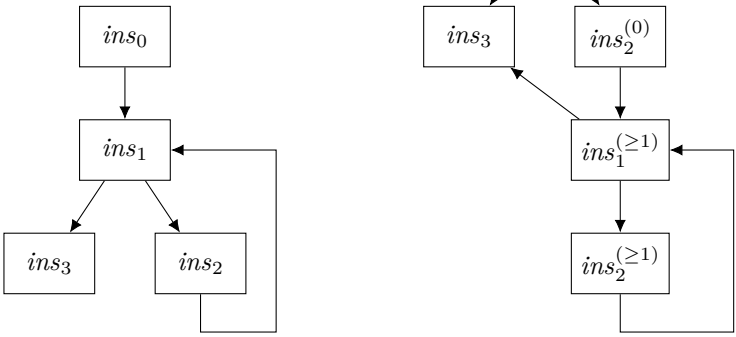
Figure 3.2: General Overview of Timing Analysis.

account. In Figure 3.2, we depict the overall approach involving *low-level* and *schedulability* analysis.

Low-level analysis computes *weight* characteristics for individual tasks. These weight characteristics, or weights for short, describe properties of all execution behaviours of the given task. Examples include the number of executed cycles or the number of memory accesses performed. The low-level analysis accounts for the effect of varying inputs and different microarchitectural states on the weights. In Section 3.3, we provide details on the techniques used in low-level analysis to compute the weights of a task.

The interface between low-level and schedulability analysis, i.e. the set of weights to be computed, depends on the chosen decomposition of the system under analysis. Some decompositions allow for a more efficient analysis while others allow for more precise results or impose fewer restrictions on the underlying hardware platform. In Section 3.4, we discuss different possible decompositions.

Based on the computed weights, the schedulability analysis computes for each task a bound on the task's response times considering the interference effect of the other tasks in a compositional way. The schedulability analysis is specific to the scheduling policy employed by the operating system task. In Section 3.5, we present the response-time analysis used for fixed-priority scheduling policies.



(a) Control-flow graph without context sensitivity.

(b) Refined control-flow graph with context sensitivity discriminating the first loop iteration.

Figure 3.3: Two possible control-flow graphs for a program with a loop.

3.3 Low-Level Analysis

Low-level analysis computes weight characteristics of a given task t. The weights characterise the execution of a single instance of t, i.e. a single execution of the underlying program p. The main task within low-level analysis is the construction of an abstract execution graph.

The program p is given as a *control-flow graph* (CFG) that describes all paths through the program during its execution. The nodes represent single instructions or basic blocks, and the edges model the control flow between these nodes. The CFG of a program can either be obtained by compilation or by reconstruction from a binary.

A CFG can be refined using *context sensitivity*. A *context* describes the circumstances under which an instruction is executed. Using contexts, the CFG can distinguish between different iterations of a loop or function calls

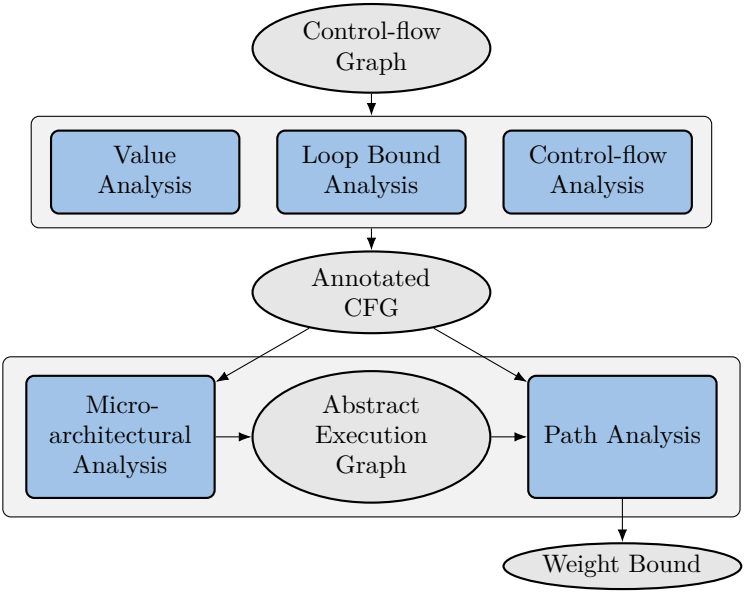


Figure 3.4: Overview of the general steps in low-level analysis.

from different call sites. For the sake of brevity, we do not provide details on the formalisation of contexts, but we refer the interested reader to the formalisation of *trace partitioning* in [Mauborgne and Rival, 2005; Rival and Mauborgne, 2007]. As an example, consider Figure 3.3.

The control-flow graph, with or without contexts, is already an abstract execution graph. However, the control-flow graph contains too little information about the program execution to estimate most weights, such as the timing. Thus, we will construct abstract execution graphs with more information. Those can be seen as enriched version of the control-flow graph.

The overall low-level analysis flow is depicted in Figure 3.4. First, we run preprocessing analyses which compute information that does not depend on the microarchitectural behaviour. These analyses include value analysis that approximates the values of registers and memory cells, and control-flow and loop analysis that approximate feasible paths through the control-flow graph. The resulting abstract execution graphs are isomorphic to

the control-flow graph, but contain additional information about possible program executions.

Second, we analyse the microarchitectural behaviour of the program using the information from the preprocessing analyses. The resulting abstract execution graph models the detailed program execution on the given hardware platform. This graph is now suited for the weight computations.

Finally, path analysis maximises weights on the abstract execution graph. As the graph is usually large, we reduce it by combining edges and nodes. The worst-case trace is found by using Integer Linear Programming (ILP). The ILP framework is well-suited to incorporate information from previous analyses as linear constraints.

While the preprocessing analyses traditionally fit the framework of static program analysis via abstract interpretation, the microarchitectural analysis is more closely related to model checking approaches. Beyer et al. [2007] present the unified *configurable program analysis* that can be tuned between precision-driven model checking techniques and efficiency-driven abstract interpretation techniques. We present their general algorithm before we show the respective instantiations for preprocessing and microarchitectural analyses.

Configurable Program Analysis

The configurable program analysis by Beyer et al. [2007] computes a set of reachable abstract configurations. We modified the original algorithm to construct abstract execution graphs, i.e. to additionally maintain the set of edges between the configurations. The result is shown in Algorithm 1.

The algorithm takes an abstraction as input given by a partially-ordered set of configurations $(\widehat{\mathcal{C}}, \sqsubseteq)$ and the respective abstract transition relation $\widehat{cycle}$. Furthermore, it takes an operator $\texttt{merge} : \widehat{\mathcal{C}} \times \widehat{\mathcal{C}} \to \widehat{\mathcal{C}}$ that is used to merge two abstract configurations into a single one. The decision which configurations to merge influences the precision of the resulting graph and the efficiency of the construction algorithm. In contrast to [Beyer et al., 2007], we require the merge operator to either not merge $\texttt{merge}(c_1, c_2) = c_2$ or to return an upper bound $\texttt{merge}(c_1, c_2) \sqsupseteq c_1, c_2$. Finally, the input $I \subseteq \widehat{\mathcal{C}}$ denotes the set of initial configurations and $F \subseteq \mathcal{E}$ the set of final events.

The algorithm starts from the initial configurations I to construct an abstract execution graph up to the occurrence of final events. During the construction, it maintains a set `worklist` of configurations that still need

Algorithm 1: Construct an abstract execution graph

Input : abstract configurations $(\widehat{\mathcal{C}}, \sqsubseteq)$ with transition relation $\widehat{cycle}$, `merge`-operator, initial configurations I, final events F
Output : abstract execution graph (V, E, I, F)

```
worklist := I
V := I
E := ∅
while worklist ≠ ∅ do
    pop c from worklist
    foreach evs, c' with ĉycle(c)(evs)(c') do
        // Explore new, non-final successor configuration
        if (c' ∉ V ∧ evs ∩ F = ∅) then
            worklist := worklist ∪ {c'}
        end
        // Add successor configuration to graph
        V := V ∪ {c'}
        E := E ∪ {(c, evs, c')}
        // Try to merge graph nodes
        foreach v ∈ V \ {c'} do
            if (c' ⊑ merge(c', v)) then
                c_new := merge(c', v)
                V := V[c_new/v][c_new/c']
                E := E[c_new/v][c_new/c']
                worklist := worklist[c_new/v][c_new/c']
                break
            end
        end
    end
end
return (V, E, I, F)
```

to be explored. In each iteration of the outermost loop, the $\widehat{cycle}$ transitions of a configuration from the worklist are explored and added to the graph. New successor configurations that have not yet encountered a final event are added to the worklist. Using the `merge`-operator, the algorithm tries to combine the successors with existing graph nodes. We denote by $V[a/b]$ that each occurrence of b in V is replaced by a.

As outlined in [Beyer et al., 2007], it is expensive to iterate over the complete set of nodes V to find merge candidates in each step. If the datastructure holding V is sorted, it might be possible to efficiently identify a small subset of V as potential merge candidates. We provide more details later in this section.

There are further extensions to Algorithm 1 to increase efficiency, which we omit in the above presentation for the sake of brevity. First, the algorithm can be enhanced to exploit cooperation as introduced in Section 2.4. Abstract execution graphs from preceding analyses or knowledge about properties of the concrete system can be used to detect infeasible cycle transitions. If every partial trace ending in (evs, c') is found to be infeasible, the algorithm does not need to explore c' further. As an example, microarchitectural analysis does not need to explore the behaviour along program paths that cannot be taken according to the preceding control-flow analysis. Second, the algorithm can account for information about domination as introduced in Section 2.5. If a cycle transition has multiple successor configurations due to non-determinism, the configurations that are dominated by others do not need to be explored. While the resulting graph is sound for weight characterisations, the use of cooperation is restricted as discussed in Section 2.5.

Preprocessing Analysis

In the first phase of low-level timing analysis, we compute information about the program that can be obtained from the behaviour at the instruction-set-architectural (ISA) level. This includes information about the values of registers and memory cells and the control-flow inside the program.

As an example, consider a value abstraction given by an abstract value domain $D^\sharp_{val}$. It compactly represents the values of registers and memory cells at a certain point of execution. The domain $D^\sharp_{val}$ usually constitutes a complete lattice with a well-defined least upper bound operator. Examples

for such value abstractions are intervals [Cousot and Cousot, 1976] and octagons [Miné, 2006].

These abstractions are defined relative to the ISA-level behaviour which itself is already an abstraction of the concrete system behaviour. The ISA-abstraction does not exactly fit into the formal framework presented in Chapter 2. The reason is that one computation step on the ISA-level approximates multiple cycles on the actual hardware level. The formal definition of this relation is not the topic of this thesis. We refer the interested reader to [Kovalev et al., 2014] that discusses such a relation in the scope of a hardware correctness proof. Note that the physical environment, which influences the values of registers and memory cells via sensor reads, is not modelled explicitly. Upon a sensor read, we can use the top element $\top \in D^\sharp_{val}$ of the abstract-value lattice to soundly model any possible value returned by the sensor.

The value analysis is performed along the paths through the control-flow graph of the given program. Let $\mathcal{L}$ denote the program locations, i.e. the nodes of the CFG. The abstract configurations are given by $\widehat{\mathcal{C}} := \mathcal{L} \times D^\sharp_{val}$. The first component hereby represents the value of the program counter, i.e. the next instruction to execute. Upon $\widehat{cycle}$, which denotes an ISA step, the program counter is adjusted according to the structure of the CFG. The abstract value state is updated according to the abstraction $D^\sharp_{val}$ and the instruction that is executed.

Value analysis is usually flow-sensitive, i.e. we distinguish value information for different program locations. Abstract configurations with different program locations are considered incomparable:

$$(pc, v) \sqsubseteq (pc, v') \Leftrightarrow v \sqsubseteq^\sharp_{val} v',$$

where $\sqsubseteq^\sharp_{val}$ denotes the partial order of $D^\sharp_{val}$.

Consequently, we define the `merge`-operator to not join the abstract value information if the program locations differ:

$$\texttt{merge}((pc, v), (pc', v')) := \begin{cases} (pc', v') & : \text{if } pc \neq pc' \\ (pc, v \sqcup^\sharp_{val} v') & : \text{otherwise} \end{cases},$$

where $\sqcup^\sharp_{val}$ denotes the least-upper-bound operator of $D^\sharp_{val}$. If additional context-sensitivity is used, different contexts are distinguished as well.

As initial configuration, we choose $I = \{(pc_0, \top)\}$ where pc_0 denotes the first instruction of the program and $\top$ denotes the top element of the

lattice $D^{\sharp}_{val}$. The abstract top element $\top$ hereby compactly represents all possible concrete configurations, i.e. all possible valuations of registers and memory cells. The final events F are the completion events of those instructions that end the program. The completion event of an instruction *ins* is emitted if the cycle transition evolves from *ins* to the next program counter value.

Microarchitectural Analysis

In this phase of the low-level timing analysis, we analyse the microarchitectural behaviour of the given program executed on one processor core of the system. This includes the behaviour of the processor core's pipeline and the caches.

An abstraction is the first input to the algorithm that constructs an abstract execution graph. Let us consider an example for a microarchitectural abstraction of system configurations:

$$\widehat{\mathcal{C}} := \widehat{Core} \times \widehat{Caches} \times \widehat{Memory}.$$

The microarchitectural abstraction abstracts from any values that are computed during program execution. Those only depend on the instruction-set-architectural behaviour and are approximated by the preceding value analysis. During the microarchitectural analysis, we use the result of the value analysis to e.g. compute the addresses accessed by a memory instruction.

The set $\widehat{Core}$ encompasses the abstract states of the processor core that executes the given program. These abstract core states comprise the current program counter and the state of the core's pipeline, i.e. which instructions occupy which pipeline stages. The state of the pipeline hereby corresponds to the pipeline state of concrete system configurations. This allows a precise modelling of the core behaviour. The program counter is updated during a $\widehat{cycle}$ transition according to the possible program paths through the control-flow graph of the given program. Upon uncertainty, e.g. multiple successor instructions due to a branch or uncertainty whether a cache hit or miss happens, the abstract configuration is split into all possible cases. This allows the core state to be updated analogously to the concrete *cycle* transition.

Compact abstract domains $\widehat{Caches}$ in the form of complete lattices for the analysis of caches are known for quite some time. An abstraction for

caches with LRU replacement policy was presented in [Alt et al., 1996]. The cache abstraction maintains over- and under-approximations of the cache contents in the form of may- and must-caches. The abstract cache is updated during a $\widehat{cycle}$ transition according to the memory blocks accessed by the instructions inside the pipeline. The transition of the processor core state itself relies on the cache abstraction to classify memory blocks as either always hit, always miss, or unknown. If an accessed block is classified as unknown, a split is performed in $\widehat{cycle}$ to explore both possible cases: cache hit and cache miss.

The abstract memory states $\widehat{Memory}$ essentially comprise the state of the memory controller. An abstract memory state is updated during a $\widehat{cycle}$ transition according to the memory operations within the core pipeline. If the state of the memory controller is kept rather concrete, the latency of a memory access is determined by this state. Another possibility is to abstract the memory state to a single value that models the remaining latency of the ongoing access. During an abstract cycle transition, the remaining latency is decremented by one. Upon a new access, the remaining latency is drawn non-deterministically among all possible latencies. To model this non-deterministic choice, the abstract configuration is split to consider all possible remaining latencies.

Note that our presented abstraction does not consider the state of the other cores of the system or the shared bus. This is due to the fact that the low-level timing analysis computes weight characteristics of a program in isolation. We consider the effects of the other cores later during schedulability analysis. However, it is possible to model these effects already during the low-level analysis. As an example, Jacobs et al. [2015] propose a low-level analysis that incorporates the effect of interference on a shared bus with round-robin arbitration. To this end, the bus interference is modelled as a non-deterministic latency prolongation of memory accesses. We will revisit this possibility when we present approaches to achieve compositionality in Chapter 5.

The microarchitectural abstraction sketched here fits the formalisation presented in Chapter 2. The local consistency of $\widehat{cycle}$ can be shown based on the local consistency of the abstract cache update.

For more detailed descriptions of abstractions for realistic microarchitectures, we refer the interested reader to [Langenbach et al., 2002; Thesing, 2004].

While the abstract cache domains with partial order $\sqsubseteq_c$ are usually complete lattices, it is not obvious how to order pipeline or memory controller states. As a consequence, the de-facto standard way [Langenbach et al., 2002; Thesing, 2004] is to keep different pipeline and memory controller states separate, i.e. to treat them as incomparable. This results in the following overall partial order $\sqsubseteq \subseteq \widehat{\mathcal{C}} \times \widehat{\mathcal{C}}$:

$$(p, c_1, m) \sqsubseteq (p, c_2, m) \Leftrightarrow c_1 \sqsubseteq_c c_2.$$

Note that the set of abstract configurations $\widehat{\mathcal{C}}$ does *not* constitute a complete lattice. As a consequence, no least-upper-bound operator exists for sets of abstract configurations. However, this does not influence the applicability of our Algorithm 1 (Configurable Program Analysis).

As initial abstract configurations I we need to conservatively consider any pipeline state where the program counter corresponds to the first instruction of the given program, any cache state, and any memory controller state. While all possible cache states can be compactly represented by the top element $\top_c$ of the abstract cache lattice, the pipeline and memory states have to be explicitly enumerated. To reduce the number of initial configurations, the concrete system can be designed such that each program starts with an empty pipeline and an idle memory controller. The final events F are the same as in the concrete case: a final events occurs when an instruction that ends the program execution leaves the pipeline.

The `merge`-operator can join abstract cache states if the remaining parts of the abstract configurations coincide

$$\texttt{merge}((p, c, m), (p', c', m')) := \begin{cases} (p', c', m') & : \text{if } p \neq p' \vee m \neq m' \\ (p, c \sqcup_c c', m) & : \text{otherwise} \end{cases},$$

where $\sqcup_c$ denotes the least-upper-bound operator of the abstract cache lattice.

As hinted in the discussion of Algorithm 1, it is expensive to search the full set V for merge candidates in each step. There are two techniques that can be used to reduce this overhead. First, we invoke the search for merge candidates only at specific points, e.g. if an instruction just left the pipeline or a memory access finished. This can reduce the number of calls to `merge` significantly at the cost of a slightly larger execution graph. Second, we keep the nodes of the graph V in a sorted datastructure. As an example,

we can sort the nodes according to the next-to-complete instruction in the respective abstract configuration. In Algorithm 1, it is sufficient to only consider nodes $v \in V$ with the same sorting key as c' as merge candidates.

Abstract Execution Graph Compression

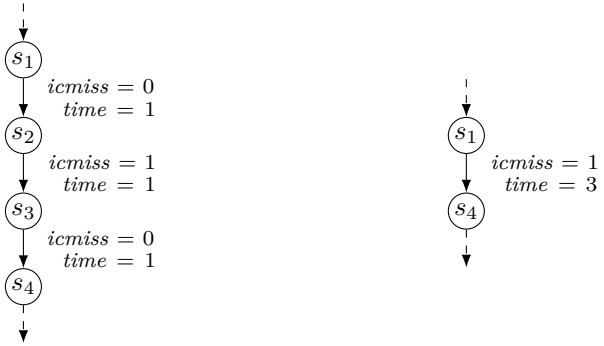
In the previous sections, we have shown how to construct an abstract execution graph at the granularity of individual processor cycles. To obtain the per-task weight characteristics needed for the schedulability analysis, we are left with finding a path through this graph that maximises certain weights.

As a first step, we transform the abstract execution graph into a form that is more suitable for the remaining analysis steps. In the transformed graph, nodes are sets of abstract configuration $\widehat{V}_p \subseteq 2^{\widehat{C}}$ with a node c in the original graph being replaced by the singleton set $\{c\}$. The edges $\widehat{E}_p \subseteq \widehat{V}_p \times \widehat{V}_p$ are no longer labelled with individual events, but with per-edge weight characteristics that count the number of occurrences of certain events. We model these by functions $w : \widehat{E}_p \to W$.

As the abstract execution graph at cycle granularity can be large, we want to reduce its size prior to the search for the longest path. Stein [2010] provides an overview of techniques to compress the abstract execution graph. Some of them preserve the weight characteristics exactly, while others trade off smaller graph size for sound but more imprecise weight characteristics. In the following, we present two of these techniques.

The first technique, termed *chain compression* by Stein, replaces a sequence of edges by a single edge. All nodes within this sequence except the start and end node are required to have a single predecessor and successor. The weights along the sequence are added up according to Equation 2.3. This compression technique preserves weight characteristics exactly, because all edges in the sequence must be taken the same number of times. For an example, consider Figure 3.5.

The second technique, termed *lossy buddy node merging* by Stein, merges two buddy nodes, i.e. nodes with the same predecessor or successor nodes, into a single node. Edges with the same source and destination are consequently merged as well. If the weights on the edges to be merged coincide, this technique preserves per-edge weight characteristics exactly. If the weights on the edges to be merged do not coincide, we maintain lower and upper bounds on the per-edge weights, i.e. the functions become



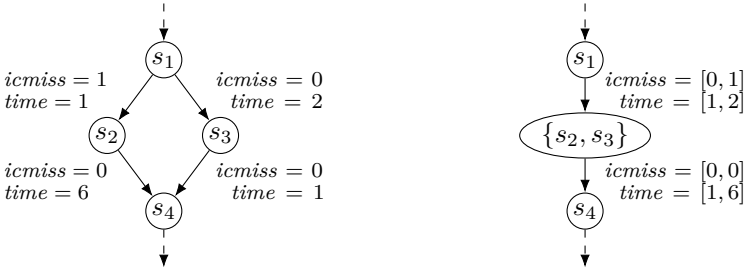
(a) Execution graph at single-cycle granularity. (b) Compressed execution graph.

Figure 3.5: Chain compression. *icmiss* denotes the number of instruction cache misses and *time* the number of elapsed cycles.

$w : \widehat{E}_p \to W \times W$. These bounds are determined using the minimum or maximum, respectively. Depending on whether a weight is maximised or minimised, the upper or lower bound is used in the later path analysis. The correctness of this compression technique follows from the fact that every path through the original graph has a corresponding path through the compressed graph. For an example, consider Figure 3.6. The example demonstrates that the compression comes at the cost of imprecision of the overall weight characteristics: the worst-case time through the depicted graph fragment increases from seven to eight cycles. The applied buddy compression opens up new possibilities for the chain compression.

The above compression techniques, especially the buddy compression, can be applied with varying degrees of aggressiveness. This can be used to trade off graph size, i.e. efficiency of the later path analysis, against precision of the obtained weight characteristics. Note that the formulation of the path analysis is independent of the compression techniques used.

There are two commonly used variants of compressed execution graphs: a state-*sensitive* and a state-*insensitive* variant. The state-insensitive variant has been used in the early work on path analysis, e.g. by Li and Malik [1995] or Theiling [2002]. This variant originates from a control-flow-graph-centric view on program verification. Consequently, the abstract execution graph is compressed such that each basic block of the program's CFG corresponds to



(a) Graph after chain compression.

(b) Graph after buddy compression.

Figure 3.6: Buddy compression. *icmiss* denotes the number of instruction cache misses and *time* the number of elapsed cycles.

a *single* weighted edge in the compressed graph. While this variant results in a compact graph and a fast path analysis, the precision of the weight characteristics deteriorates due to infeasible paths.

Matthies [2006] and Stein [2010] use a state-sensitive variant which they term "prediction-file based". This variant explicitly keeps the abstract configurations that correspond to basic block transitions separate. The edges and nodes in between are compressed using the above methods. Depending on the later path analysis, it can be useful to restrict the buddy compression such that certain weights stay precise. We will see examples in the next section.

Path Analysis

The path analysis takes a possibly compressed abstract execution graph and maximises (minimises) a weight on any path through the graph. Furthermore, the path analysis implements cooperation between different abstractions as outlined in Section 2.4 (Cooperation). The information of other abstractions helps to identify infeasible paths that do not need to be considered.

An explicit consideration of all (feasible) paths through the graph will be expensive. Instead of an actual worst-case *path*, we are interested in the worst-case *weight*. This observation leads to the *implicit path enumeration technique* introduced by Li and Malik [1995]. They compute how often each

edge in the graph is taken in the worst case ignoring the order of edges in a path.

Li and Malik chose to encode the implicit path enumeration as an *integer linear program* (ILP) for which good solvers exist. The modularity of the ILP encoding enables straightforward cooperation if additional information can be described in the form of linear constraints. The ideas of Li and Malik have been used and refined many times [Theiling, 2002; Stein, 2010] and thus establish today's de-facto standard approach to path analysis. In the following, we present how to encode an implicit path enumeration for our abstract execution graph as integer linear program.

We use integer variables x_e that determine how often edge e in the graph is taken in the worst case. The objective to maximise a weight, e.g. the execution time, reads as

$$\max \sum_{e \in \widehat{E}_p} time^{ub}(e) \cdot x_e \,.$$

Note that due to the possible graph compression, we use the per-edge upper bound of the time weight, denoted by $time^{ub}(e)$. The final objective value obtained from an ILP solver is a sound upper bound on the execution times of any path through the graph. From a valuation of the variables x_e, it is often possible to reconstruct a worst-case path.

The above maximisation problem is subject to constraints. First, constraints encode the structure of the abstract execution graph. For convenience, we add a special vertex v_0 to the graph. We add edges from v_0 to each initial node and edges from each final node to v_0.

Every path through the execution graph satisfies the conservation of flow. Each time a node in the execution graph is entered, the node is left again:

$$\forall v \in \widehat{V}_p. \sum_{e=(.,v) \in \widehat{E}_p} x_e = \sum_{e=(v,.) \in \widehat{E}_p} x_e \,.$$

As we are interested in the weight characteristics of a single program execution, only one edge to an initial node is taken:

$$\sum_{e=(v_0,.) \in \widehat{E}_p} x_e = 1.$$

If a function is called from multiple call sites, there are edges from each call site to the function start and back from the function end. Naturally, if

a function is called from a call site, the function returns to that call site again:

$$\forall \text{function } f, \text{call site } i.\ \sum_{e \in \widehat{E}_p} \mathit{call}_{i,f}(e) \cdot x_e = \sum_{e \in \widehat{E}_p} \mathit{return}_{i,f}(e) \cdot x_e\,.$$

The binary weight $\mathit{call}_{i,f}$ describes whether instruction i calls function f on a given edge. Analogously, the binary weight $\mathit{return}_{i,f}$ describes whether function f returns to the instruction subsequent to i on a given edge. These weights are kept precise during the graph compression.

Second, constraints exclude infeasible paths due to information from other abstractions or knowledge about the system. Thus the set of constraints depends on the actual system and the other analyses being used. In the following, we present a collection of constraints that demonstrates the variety of application scenarios.

Li and Malik already considered loop bound information, i.e. information about how often each loop in the given program can iterate at most. This information is either provided by control-flow analyses [Ermedahl et al., 2007; Cullmann and Martin, 2007] or by user annotations. Without upper loop bounds, the objective of the ILP would become infinite for programs containing loops. The generated loop constraints enforce that a loop can iterate at most *loopbound* many times for each entrance of the loop:

$$\forall \text{loop } l.\ \sum_{e \in \widehat{E}_p} \mathit{backedge}_l(e) \cdot x_e \leq \mathit{loopbound} \cdot \sum_{e \in \widehat{E}_p} \mathit{enter}_l(e) \cdot x_e\,.$$

The weight $\mathit{backedge}_l(e)$ describes how often a backedge of loop l in the control-flow graph has been taken on execution-graph edge e. The weight $\mathit{enter}_l(e)$ describes how often loop l has been entered from the outside on execution-graph edge e. For precision reasons, these events are usually binary and kept precise during graph compression. Note that for programs with recursive functions, the recursion depth can be bounded analogously to the loop constraints.

Additional control-flow properties, often called flow facts, can often be expressed as linear constraints as well. An example is the infeasibility of certain paths through consecutive conditionals with contradicting conditions. Raymond [2014] has investigated the limits of expressing such control-flow infeasible paths as linear constraints without further graph transformations.

Besides abstractions that focus on control-flow properties, there are cooperating abstractions that concentrate on microarchitectural events. A commonly known example is cache persistence analysis [Cullmann, 2013]. A memory block is called *persistent*, if it stays in the cache once it has been loaded. Persistence information is usually available within a certain *scope*, i.e. a portion of the program execution such as the iterations of a loop. A memory block b which is persistent in the instruction cache within scope $S \subseteq \widehat{E}_p$ can thus cause at most one miss for each entrance of S:

$$\sum_{e \in S} icmiss_b^{lb}(e) \cdot x_e \leq \sum_{e \in \widehat{E}_p} enter_S^{ub}(e) \cdot x_e \,.$$

The weight $icmiss_b^{lb}(e)$ is a lower bound on the number of instruction cache misses when accessing memory block b on edge e. The weight $enter_S^{ub}(e)$ is an upper bound on the number of scope entrances on edge e. If the above constraint is violated for an abstract trace, it is guaranteed that every represented concrete trace exhibits at least two misses for a scope entrance which is infeasible. For a formal argument when to use lower and upper bounds on per-edge weights, we refer the interested reader to [Jacobs, 2018].

For a system with write-back data caches, it is often hard to locally decide whether a cache miss causes a write-back or not. However, an analysis can compute an overall upper bound on the number of possible write-backs by counting the number of dirtifying stores. Blaß et al. [2017] call a store *dirtifying* if it might turn a previously clean cache line dirty, thus causing a write back in the future. This information can be encoded via the following linear constraint:

$$\sum_{e \in \widehat{E}_p} writeback^{lb}(e) \cdot x_e \leq \sum_{e \in \widehat{E}_p} dfstore^{ub}(e) \cdot x_e \,.$$

The weight $writeback^{lb}(e)$ is a lower bound on the number of write backs that happen on edge e. The weight $dfstore^{ub}(e)$ is an upper bound on the number of dirtifying stores on edge e.

In a system with Dynamic Random-Access Memory (DRAM), periodic refreshes are necessary to keep the memory content stable. It is hard to determine the individual accesses that are delayed by those refreshes. However, using system knowledge such as the minimum time between the

arrival of two refreshes, a global upper bound on the number of refreshes started during program execution can be obtained:

$$(\mathit{maxRefreshes} - 1) \cdot \mathit{refreshInterArrivalTime} + 1 \leq \sum_{e \in \widehat{E}_p} \mathit{time}^{ub}(e) \cdot x_e \,.$$

This upper bound is then used to restrict the frequency of edges that correspond to refreshes:

$$\sum_{e \in \widehat{E}_p} \mathit{refresh}^{lb}(e) \cdot x_e \leq \mathit{maxRefreshes}.$$

The weight $\mathit{refresh}^{lb}(e)$ is a lower bound on the number of refreshes that might happen on edge e.

The presented microarchitectural abstraction has been designed for programs run in isolation, i.e. without interference of other tasks on the shared resources. Instead of accounting for the interference effects during schedulability analysis, it is also possible to account for the effects of interference during low-level analysis at higher analysis cost. Jacobs et al. [2015] consider the timing effects of shared-bus blocking within the low-level analysis by modelling interference as non-deterministic delays. Additionally, the maximal possible interference I of the co-running cores on the execution of the program p has been determined. Any abstract trace that encounters more interference than I cycles is thus infeasible. The resulting constraint reads as

$$\sum_{e \in \widehat{E}_p} \mathit{blocked}^{lb}(e) \cdot x_e \leq I.$$

The weight $\mathit{blocked}^{lb}(e)$ is a lower bound on the number of blocked cycles that happen on edge e.

This section ends the discussion of how state-of-the-art low-level analysis is performed. It includes the derivation of instruction-set-level program properties and the microarchitectural analysis that constructs an abstract execution graph. The path analysis computes weight characteristics from the compressed execution graph using additional information to exclude infeasible paths. These weight characteristics serve as inputs to the subsequent schedulability analysis.

3.4 Scheduling Interface and Compositionality

In the early 70s, a simple interface between low-level analysis and high-level schedulability analysis emerged [Liu and Layland, 1973; Muntz and Coffman Jr., 1970]. A low-level analysis hereby computes per-task weight characteristics that capture the execution time of a task on the underlying processor in isolation. This characteristic of task t_i is denoted by w_i [Muntz and Coffman Jr., 1970] or, more commonly in use, C_i [Liu and Layland, 1973]. Based on these characteristics, either a schedule is calculated or the existence of a schedule is checked.

The system under analysis operates in a timely manner if the worst-case response time of every task as defined in Section 3.2 does not exceed its deadline. To prove the timeliness of the system, it is sufficient to show that an upper bound on the response times of each task is within the respective deadline. Upper bounds on the response times of tasks are obtained by additive operations among the per-task weights. As an example, the response time of a task t_2 that is preempted twice by a task t_1 is bounded by

$$WCRT_2 \leq C_2 + 2 \cdot C_1.$$

Thus, the associated schedulability analysis requires *timing compositionality* to be sound. To satisfy the decomposition into per-task execution times, all schedulability analyses that build upon this interface make simplifying assumptions. Muntz and Coffman Jr. [1970] argue that the preemption and task-switching costs are negligible in their setting and consequently assume them to be zero. Liu and Layland [1973] assume that the weights C_i take the task-switching and preemption costs into account—offloading the responsibility on to the low-level analysis.

Preemptions In modern embedded systems with complex microprocessors, the above assumptions become problematic. Stärner and Asplund [2004] measure the effect of *cache-related preemption delay*, i.e. the cost to reload cache lines evicted due to preemption, on the execution time. They conclude that the preemption cost can be significant and needs to be considered in a timing analysis. More importantly, the cache-related preemption delay depends on the actual task schedule, i.e. which tasks are preempting and how often. Thus, offloading the computation of preemption cost on to the low-level analysis is not an option without giving up the useful separation

of concerns. Even if a schedule-independent preemption cost bound can be computed, it will lead to overly pessimistic results. The above interface should be refined in order to keep the separation of concerns between per-task low-level analysis and schedulability analysis, and to enable a precise overall timing analysis.

Busquets-Mataix et al. [1996] propose to account for the cache-related preemption delay of a single preemption by adding the cost to refill either the entire cache or only the cache lines evicted by the preempting task. Later, Tomiyama and Dutt [2000] show how to actually compute the worst-case number of cache lines accessed by a task. The underlying scheduling interface is the following:

- C_i is the execution time of task t_i in isolation assuming the absence of preemptions, and
- ECB_i is the number of evicting cache blocks, i.e. the number of cache lines accessed by task t_i.

If a task t_j preempts a task t_i, ECB_j is used to bound the cache-related preemption delay. As an example, consider task t_2 preempted by t_1 on a system with direct-mapped cache:

$$WCRT_2 \leq C_2 + C_1 + ECB_1 \cdot brt,$$

where the block reload time (brt) denotes the time needed to reload a cache line from memory.

Lee et al. [1996] extend the scheduling interface by the number of useful cache blocks UCB_i of task t_i. The term "useful cache block" has first been used by Lim et al. [1994]: A cache line is considered useful at a program point of task t_i if it might be cached at this point and might be reused later. The worst-case number of useful cache blocks is obtained by taking the maximum over all program points. If a task t_i is preempted by another task, only useful blocks can cause preemption-induced reloads. Thus, UCB_i is used to bound the cache-related preemption delay. As an example, consider task t_2 preempted by t_1 on a system with direct-mapped cache:

$$WCRT_2 \leq C_2 + C_1 + UCB_2 \cdot brt.$$

Building up on this new interface, researchers have proposed further refinements, e.g. *definitely-cached* useful cache blocks, and more powerful

schedulability analyses. An overview of the work in this area can be found in [Altmeyer, 2013].

The cache-related preemption delay is not the only cost associated with preemptions. A preemption is realised by the scheduler of the operating system that is invoked via periodic timer interrupts. Thus, the preemption delay also encompasses the *pipeline-related interrupt cost* as well as the execution time of the interrupt service routine (ISR) which in turn invokes the scheduler. The pipeline-related interrupt cost can be hard or impossible to determine, especially when the pipeline features domino effects as shown in [Lundqvist and Stenström, 1999]. In Section 5.4 (Compositionality by Sound Penalty), we bound this cost for our strictly in-order pipeline which we propose in Section 4.5. The ISR can be modelled as an individual task with its own weights such as its execution time C_{ISR}. The ISR has the highest priority among all tasks and is invoked periodically with the period given by the external timer component. A sound high-level schedulability analysis has to account for the effects of the ISR, including the cache-related preemption delay and the interrupt-related pipeline cost. For more details on how to model the operating system in schedulability analysis, we refer to the dissertation of Schneider [2003].

The extension of the interface between low-level and high-level analysis comes at a cost which is neglected in the literature. For an overall soundness guarantee, the authors of schedulability analyses need to formally prove that the proposed interface, i.e. the set of weight characteristics, together with a combination operator actually forms a *decomposition* of the response time in the sense of Section 2.6 (Compositionality). This involves the choice of the penalty *brt*. Choosing the memory latency as the obvious penalty might not be sufficient due to anomalous timing effects inside the system. In general, it is an open question how to compute this penalty or how to prove that a given penalty is sufficient. The, often "hidden", compositionality assumption is an actual show-stopper for the applicability of these schedulability analyses to modern systems. In Chapter 5 (Achieving Timing Compositionality), we approach this problem and propose solutions.

Multi-Core Processors With the advent of multi-core processors, the response time of a task is additionally influenced by concurrent tasks on other cores due to the competition for shared resources such as buses and memory. The same questions as in the case of the cache-related preemption

delay arise. The interference that a task experiences depends on the tasks concurrently executed, i.e. the actual task schedule. Incorporating the maximal possible interference effects into the weight C_i during the low-level analysis is likely to lead to overly pessimistic results [Abel et al., 2013].

Recently, Altmeyer et al. [2015] have presented an interface extension to additionally account for shared-bus interference within schedulability analysis. The weight C_i captures the execution time of task t_i in isolation assuming the absence of preemptions and shared resource interference. Besides the weights ECB_i and UCB_i to characterise the cache-related preemption delay, they introduce the memory demand MD_i, i.e. the maximal number of accesses of task t_i to the shared bus. Based on these weight characteristics and the memory latency, they compute the effect of shared-bus interference for different arbitration policies. The calculated effect is added to the execution times in a similar fashion as above to obtain the worst-case response time.

The proof obligations that arise w.r.t. compositionality are similar to the obligations associated with the cache-related preemption delay. Thus, without modifications, the approach in [Altmeyer et al., 2015] is *not sound* to use for most modern systems. The techniques we present in Chapter 5 (Achieving Timing Compositionality), however, enable its application for any multi-core system with shared resource interference.

3.5 Schedulability Analysis

In the previous sections, we have presented a generic low-level analysis to derive weight characteristics of individual program runs in isolation. Furthermore, we have shown possible interfaces between low-level and schedulability analysis, i.e. the set of weight characteristics to be passed. Finally, we want to shed some light on the schedulability analysis itself.

Schedulability analysis determines whether a *set of tasks* with weight characteristics according to the above interface is schedulable with a given *scheduling policy*. The task set is schedulable if all deadlines of all tasks are met by the schedule obeying the given policy.

A *task* is a program with additional scheduling parameters. The scheduling parameters describe how dynamic instances of the task, called *jobs*, are released and which timing requirements must be met. We distinguish three categories of tasks. *Periodic* tasks release new jobs separated by a constant

time interval, the period T. A periodic task specifies a relative deadline D, i.e. the deadline of a job is given relative to its release. The deadlines are usually implicitly given or constrained by the period, i.e. either $D = T$ or $D \leq T$. *Aperiodic* tasks release new jobs at irregular time intervals. A special type of aperiodic tasks are *sporadic* tasks. A sporadic task is characterised by a relative deadline and a minimum inter-arrival time between consecutive jobs. Scheduling parameters such as periods, inter-arrival times, and deadlines are dictated by the physical environment and provided by the system engineer. As an example, the period of a task could be determined by the sampling frequency of a sensor which must be twice the highest input frequency to prevent aliasing. Timing requirements, i.e. the periods and deadlines of tasks, might also originate from the discretisation of continuous closed-loop PID controllers as described by Åström and Wittenmark [1996]. To summarise, a task description is given by the weight characteristics of the underlying program and the set of scheduling parameters.

The scheduling policy decides which job to execute among all currently available jobs. We distinguish *preemptive* scheduling, where a job's execution can get interrupted by another job, and *non-preemptive* scheduling, where a job's execution is atomic. The scheduler that implements a policy is invoked either directly by a job or at regular intervals by hardware timer interrupts. The time between two interrupts, called a time unit, is usually in the range of micro- to milliseconds.

In hard real-time systems, priority-driven schedulers as presented by Liu and Layland [1973] are common. Liu and Layland distinguish policies that dynamically assign priorities to jobs (e.g. earliest deadline first scheduling) from policies that use statically fixed task-level priorities (e.g. rate monotonic scheduling).

Scheduling a set of tasks on a multi-core or multiprocessor offers an additional degree of freedom: migration. The possibilities range from *global* scheduling, where jobs can migrate freely among the available cores, to *partitioned* scheduling, where all jobs of a task are mapped to a fixed core. A survey of multiprocessor scheduling including a recap of scheduling notation can be found in [Davis and Burns, 2011].

Given a set of tasks and a scheduling policy, schedulability analysis checks whether all jobs of all tasks will meet their deadlines without constructing the actual schedule. There are sufficient tests based on the notion of processor utilisation $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$, e.g. a set of tasks is guaranteed to be schedulable by fixed-priority scheduling on a uniprocessor if $U \leq \ln 2$.

A more precise schedulability analysis is the *response-time analysis* that computes worst-case bounds on the response times of each task.

Example: Response-Time Analysis

For the sake of brevity, we limit ourselves in the following to partitioned, fixed-priority, preemptive scheduling of a set of periodic tasks with implicit deadlines. The tasks execute on a multi-core system as depicted in Section 3.1 with direct-mapped caches. We present a response-time analysis similar to the one proposed by Altmeyer et al. [2015].

We are given a set of periodic tasks t_i with respective period T_i. We assume the tasks to have globally unique fixed priorities. As we consider a partitioned setting, function c maps each task to the core it is executed on. We use the functions $lp(i), hp(i), hep(i)$ to denote the set of tasks on core $c(i)$ with lower, higher, or higher or equal priority than task t_i.

Similar to the proposed interfaces in Section 3.4, a task t_i has the following weight characteristics:

- C_i, the computation time of task t_i when executed in isolation,
- MD_i, the memory demand of task t_i, i.e. the number of memory accesses when executed in isolation,
- $ECB_i^{data/ins}$, the set of all cache sets with evicting cache blocks, and
- $UCB_i^{data/ins}$, the set of all cache sets with useful cache blocks.

Note that the characterisations of the evicting and useful cache blocks used here are more informative than their overall number as used in Section 3.4.

The response-time analysis calculates an upper bound on the response times of each task. The timeliness of the overall system is guaranteed if the upper bound—and thus the worst-case response time—of each task is within the respective deadline.

Response Time The upper bound R_i on the response times of task t_i is derived from the computation time in isolation and the interference effects on all shared resources:

$$R_i = C_i + Core_i(R_i) + Cache_i(R_i) \cdot ml + Bus_i(R_i) \cdot ml + Dram_i(R_i) \cdot rl,$$

where *ml* (*rl*) denotes the access (refresh) latency of the main memory. The interference on the processor cores and the local caches is due to preemption, while the interference on the shared bus is caused by tasks running on other cores.

The amount of interference depends on the time available to generate the interference, i.e. it depends on the response time itself. Thus, the response time equation is recursive and the calculation of its least solution requires a fixed-point iteration.

Note the inherent *compositionality* assumption, that the individual interference contributions can be added up to a sound response time. The calculation of the individual contributions based on the provided per-task weight characteristics is discussed in the following.

Interference on the Core The response time of a task t_i is prolonged by the computation of higher-priority tasks that preempt t_i. This has been addressed in the early work by Liu and Layland [1973]. They showed that the *critical instant* for t_i which leads to the longest response time occurs when all tasks with higher priority are released simultaneously with t_i. As a consequence, the interference on the processor core can be estimated by

$$Core_i(t) = \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j.$$

The fraction $\lceil \frac{t}{T_j} \rceil$ describes how often a task t_j of higher priority can preempt t_i during time interval t.

Interference on the Local Caches As a first step, we calculate the cache-related preemption delay $\gamma_{i,j}$ of a task t_j preempting t_i *once*. An approach to determine $\gamma_{i,j}$ from the UCB and ECB characteristics is the ECB-Union approach proposed in [Altmeyer et al., 2011]. Task t_k denotes the task that is directly preempted by t_j. Due to nested preemptions, evicting cache blocks of t_j and all tasks of higher priority can evict useful cache blocks of t_k. We obtain the cache-related preemption delay by considering all possible t_k:

$$\gamma_{i,j} = \max_{k \in \mathrm{hep}(i) \cap \mathrm{lp}(j)} \left| UCB_k \cap \bigcup_{h \in \mathrm{hep}(j)} ECB_h \right|.$$

The delay $\gamma_{i,j}$ is calculated for the instruction and the data cache, respectively.

Similar to the interference on the core, we finally obtain the interference on the local caches due to preemptions by

$$Cache_i(t) = \sum_{j \in \mathrm{hp}(i)} \left\lceil \frac{t}{T_j} \right\rceil \cdot \left(\gamma_{i,j}^{data} + \gamma_{i,j}^{ins} \right).$$

Interference on the Shared Bus First, we estimate the maximal number of bus accesses started by core $c(i)$ during the execution of task t_i. Task t_i can be preempted by tasks t_j of higher priority. Each such task t_j accesses the memory MD_j times and causes additional accesses due to the cache-related preemption delay $\gamma_{i,j}$. The critical instant occurs when the tasks of higher priority are released simultaneously:

$$S_i(t) = \sum_{j \in \mathrm{hep}(i)} \left\lceil \frac{t}{T_j} \right\rceil \cdot \left(MD_j + \gamma_{i,j}^{ins} + \gamma_{i,j}^{data} \right).$$

Second, we estimate the maximal number of bus accesses started on concurrent core $c(k)$ by tasks of higher or equal priority than t_k during any time interval t. The critical instant does not occur when the tasks on core $c(k)$ are released simultaneously to t_i. Rather, the critical instant occurs when all tasks with higher or equal priority than t_k start with their maximal number of accesses at the end of their respective executions. In the worst case, we thus need to consider the number of bus accesses of a task t_j for $\lceil \frac{t}{T_j} \rceil + 1$ times:

$$A_k(t) = \sum_{j \in \mathrm{hep}(k)} \left(\left\lceil \frac{t}{T_j} \right\rceil + 1 \right) \cdot \left(MD_j + \gamma_{k,j}^{ins} + \gamma_{k,j}^{data} \right).$$

A more precise, but also more complicated formulation can be found in [Altmeyer et al., 2015].

For a shared bus with round-robin arbitration, we can derive the amount of interference as described in [Altmeyer et al., 2015]:

$$Bus_i(t) = 1 + \sum_{l_y \ s.t. \ y \neq c(i)} \min\{A_{l_y}(t), S_i(t)\},$$

where l_y denotes the index of the lowest-priority task on core y. Each access on core $c(i)$ can be blocked by one access of each concurrent core y. As round-robin does not care about priorities, even a low-priority task on core y can cause an access of t_i to be blocked. In addition, t_i can be delayed initially by a single ongoing (non-preemptive) memory access of a lower-priority task on core $c(i)$.

Interference on the Memory For certain types of main memory, such as dynamic random-access memory, there is additional interference that affects the response time of a task. The interference on the dynamic random-access memory is caused by the memory controller issuing refreshes to each row at least once per time interval T_{refr} to keep the memory content. The amount of interference depends on the policy of the controller to schedule refreshes. One possible policy is to evenly distribute the refreshes to each row within T_{refr}, i.e. a refresh occurs every $\frac{T_{refr}}{\#rows}$ time units. A bound on the number of refreshes that can happen during time t is determined by

$$Dram_i(t) = \left\lceil \frac{t \cdot \#rows}{T_{refr}} \right\rceil .$$

Note that the number of refreshes is calculated from the characteristics of the used memory controller and thus does not require an additional preceding low-level analysis. A refined formula, also extended to other refresh policies, can be found in [Altmeyer et al., 2015].

3.6 Implementation/Tool Support

We implemented a low-level analysis tool following the scheme sketched in Figure 3.4 in Section 3.3 (Low-Level Analysis). In this section, we provide details on the tool. Our tool, called LLVMTA, is based on the LLVM compiler infrastructure [Lattner and Adve, 2004]. LLVMTA is hooked into the common LLVM compilation flow as depicted in Figure 3.7.

Overall Tool Architecture Given a C program, the compiler frontend CLANG[1] translates the program into the LLVM intermediate representation. After an optional optimisation phase (OPT), the program is further

[1] `https://clang.llvm.org`

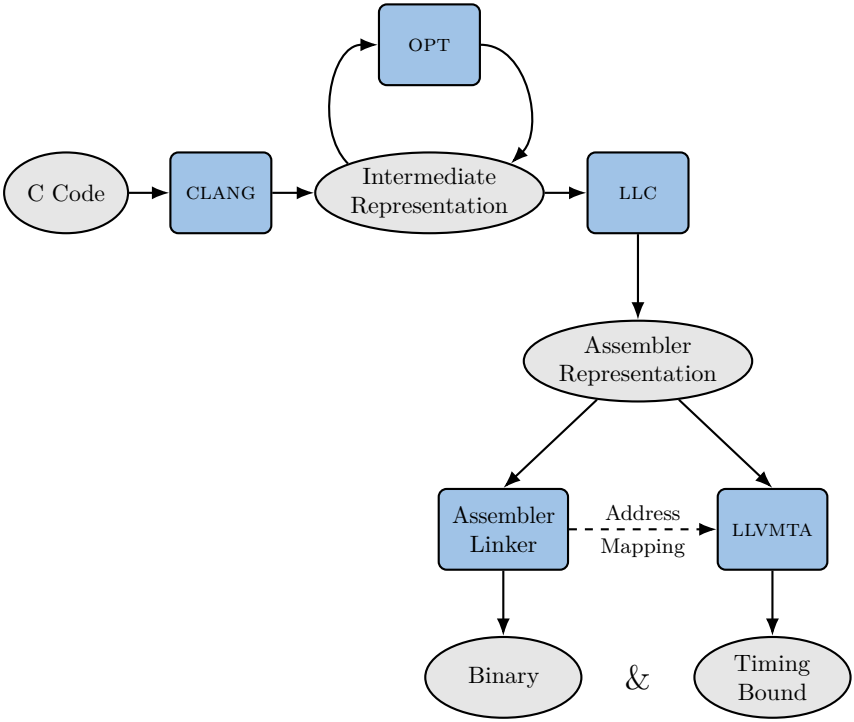


Figure 3.7: Overview of the common LLVM compilation flow (CLANG, OPT, LLC) including the integration of our low-level analysis tool LLVMTA.

translated to the assembler code (LLC) which results in the final binary after the linking step. Our analyses are implemented on the final assembler representation in the LLVM backend which is the representation closest to the machine level. The timing bound determined by LLVMTA is valid for the resulting binary, i.e. it will change accordingly if the binary changes, e.g. due to different compiler optimisations.

The integration of low-level timing analysis and compilation offers several advantages. First, no control-flow reconstruction of the binary is required because control-flow elements such as functions, basic blocks, and loops are provided by the prior compilation step. Second, the low-level analysis in the backend can make use of (high-level) information obtained at earlier stages and maintained during compilation. On the downside, the analysis requires the high-level source program, for example given in C, and provides timing estimates only for the binary produced by the specific compiler. Furthermore, the addresses of the instructions and the static data are only known after the linking step and need to be fed back to the low-level analysis.

LLVMTA To obtain precise results, we have implemented *context-sensitive* analyses, i.e. the analyses distinguish different contexts that influence the execution behaviour. As an example, the execution behaviour of the first iteration of a loop usually differs from the behaviour of later iterations because the caches are being filled during the first iteration. To establish a context-sensitive analysis framework, we implemented trace partitioning [Mauborgne and Rival, 2005] on the final assembler representation in the LLVM backend. Context sensitivity is achieved by partitioning the set of execution traces according to some predicate on traces. We implemented predicates to discriminate different iterations of a loop, as well as different call sites of a function. The degree of context sensitivity, i.e. the number and size of these predicates, is an analysis parameter.

Based on our context-sensitive analysis framework, we have implemented a value analysis that tracks constant values of registers and memory cells. This value information is used to derive address information for data accesses. Despite the simplicity of the analysis domain, it is sufficient to precisely analyse stack-relative accesses. For accesses to globally defined objects such as global arrays, our tool uses information provided by the compiler to determine the range of possible addresses.

In order to derive loop bounds, we use the LLVM-internal scalar evolution analysis that provides (an upper bound on) the iteration count of loops in their intermediate representation. Our tool additionally matches loops in the assembler representation to loops in intermediate representation in order to automatically obtain upper loop bounds on the assembler level. Manual loop annotations can be provided by the user for loops with complex iteration patterns. The scalar evolution analysis, originally based on [Bachmann et al., 1994] and extended in [Calman and Zhu, 2010], computes a closed form expression to describe how the values of variables evolve within a single loop iteration. These expressions are used to derive upper loop bounds, either in the form of numeric values or symbolic expressions w.r.t. the function parameters.

Our tool supports the analysis of different generic hardware platforms rather than proprietary industrial platforms. This is sufficient to evaluate the general concepts used in timing analysis and takes significantly less effort to implement. We model textbook pipelines (cf. [Hennessy and Patterson, 2012]) with in-order, strictly in-order, and out-of-order execution. The microarchitectural analysis supports fast local scratchpad memory, as well as caches with least-recently-used replacement policy and both write-through and write-back policy. We have implemented must, may, and persistence cache analysis. As background memory, the tool supports fixed-latency memory as well as dynamic random-access memory with a closed-page controller and distributed refreshes.

LLVMTA implements the *fast-forwarding* technique presented in [Jacobs et al., 2015] to increase the performance of the microarchitectural analysis. This optimisation exploits the fact that pipelines tend to *converge* while waiting for memory, i.e. the pipeline cannot advance further until the current memory request is finished. Once converged, the (abstract) state of the pipeline stays the same as long as the memory is busy. The microarchitectural analysis detects whether the pipeline state has converged and fast-forwards the subsequent execution to the point at which the memory is no longer busy. For more details, we refer to the explanations in [Jacobs et al., 2015].

The abstract execution graph produced by the microarchitectural analysis is compressed afterwards. LLVMTA supports two different levels of compression. Either all start and end nodes within a basic block are kept separate to allow for a precise path analysis, or the graph is compressed into a single edge per basic block to allow for an efficient path analysis. Our tool sup-

ports multiple solvers to solve the ILP formulation resulting from the path analysis, including the commercial tools IBM ILOG CPLEX Optimization Studio[2] and Gurobi Optimizer[3] that exhibit the best performance [Meindl and Templ, 2012].

Last but not least, LLVMTA supports compositional analysis approaches, i.e. several weights can be chosen for maximisation such as useful cache blocks or accesses to the shared bus. In particular, our tool can sample interference response curves and calculate compositional base bounds as introduced in Chapter 5 (Achieving Timing Compositionality).

[2] https://www.ibm.com/us-en/marketplace/ibm-ilog-cplex
[3] https://www.gurobi.com

Chapter 4

Progress-based Abstraction

Abstractions for caches and values processed in the processor pipeline are well-known in the literature [Alt et al., 1996; Cousot and Cousot, 1977]. However, no abstractions for the control of processor pipelines have been found yet.

Modern processors employ complex pipelines with a large space of concrete pipeline (control) states. The missing abstraction is an obstacle to an efficient microarchitectural analysis because the analysis needs to explicitly explore the reachable *concrete* pipeline states. Furthermore, the analysis of the pipeline control cannot cope implicitly with uncertain information provided by other analyses such as the cache analysis. As described in Section 3.3 (Microarchitectural Analysis), the pipeline control state needs to be split to explicitly explore the concrete possibilities permitted by the uncertain information. Due to the complexity of the pipeline behaviour, the pipeline analysis is prone to timing anomalies (Section 2.5). As a consequence, a sound analysis is required to explore *all* possibilities permitted by uncertain information. The huge size of the explored state space makes the pipeline analysis the most expensive part of the low-level analysis.

In 2006, Li et al. have introduced a new type of microarchitectural analysis for out-of-order pipelines. For each program instruction, they compute the time span in which the instruction might occupy a certain pipeline stage. Their presented algorithm does not rely on explicitly enumerating reachable pipeline states. However, there is no formal reasoning about the correctness of this approach.

In [Hahn et al., 2015a], we have proposed an approach to abstract the pipeline control with a similar basic idea as [Li et al., 2006]. It is based on the notion of pipeline *progress* describing how far individual instructions have advanced in the pipeline towards their completion. Intuitively,

pipeline states should be ordered according to the progress of the program instructions. A pipeline state p with less progress than a state p' should subsume p' as it should not finish earlier. Two arbitrary pipeline states can be joined to a single state by taking the *minimal progress* of the two.

In the following, we provide the formal background to reason about pipeline abstractions based on the notion of progress. For a given complex processor pipeline, the design of a correct *and* useful abstract transformer turns out to be very difficult in general. However, we show that the concrete transformer constitutes an abstract transformer in case it behaves monotonically w.r.t. the progress order.

Unfortunately, even a conventional in-order pipeline does *not* behave *monotonically* and thus prevents this straightforward use of the progress-based abstraction. In Section 4.5, we present a modified in-order pipeline design which we term *strictly in-order* and which has first been sketched in [Hahn et al., 2015a]. We prove the monotonicity of its cycle behaviour as well as interesting advanced properties such as timing compositionality (in Chapter 5) and the absence of timing anomalies (in Section 4.7). These properties enable an efficient timing verification and thus make the strictly in-order pipeline well-suited for the use in hard real-time systems. We conclude with an outlook on how to increase the processor performance while preserving monotonicity.

4.1 Formalisation of Progress-based Abstraction

The state-of-the-art microarchitectural abstraction described in Section 3.3 keeps the processor control state *concrete*. In particular, an abstract configuration is never uncertain about the progress of program instructions within the pipeline. This fact manifests itself in the definition of local consistency and the concretisation function of traces as presented in Section 2.2 (Abstraction): The occurrence of events in the abstract and concrete cycle behaviour is synchronised. An abstract cycle transition emits an event if and only if the represented concrete cycle transition emits this event.

In contrast, a progress-based abstract configuration describes multiple concrete configurations with *different* levels of progress of the instructions within the pipeline. As a consequence, we need to generalise the notion of local consistency as well as the concretisation function of traces. Before we

provide the formal background of the progress-based pipeline abstraction, we shed some light on the structure of the concrete pipeline domain.

Concrete Pipeline Domain

A machine program comprises a set of static assembler instructions. A concrete execution of a program on a specific input generates a sequence of dynamic instances of the program's instructions that is executed within the processor pipeline. As an example, the execution of a static instruction within a loop will generate a new dynamic instance of this instruction in each loop iteration. In the following, we consider a single fixed sequence of dynamic instruction instances $ins_0 ins_1 ins_2 \ldots$ that arises during the execution of a specific program for a specific input. We will outline in Section 4.2 how to deal with multiple instruction sequences, e.g. due to unknown inputs, during analysis. We denote the set of dynamic instruction instances within the sequence by $\mathcal{I}_d = \{ins_0, ins_1, \ldots\}$. Dynamic instruction instances are totally ordered by their position within the sequence, i.e. $ins_n < ins_m$ if and only if $n < m$.

The state of the pipeline control is determined by the *progress* that the instruction instances of $\mathcal{I}_d$ have made within the pipeline. An instruction instance has either been finished already, resides currently in a certain pipeline stage, or has not entered the pipeline yet. The definition of the possible levels of progress, denoted by P, depends on the actual pipeline construction. In Section 4.3 we define the notion of progress for the five-stage in-order pipeline described in Appendix A.1.

To summarise, as the set of concrete configurations we choose

$$\mathcal{C} := \mathcal{I}_d \to P,$$

where P denotes the partially-ordered set of possible progress of an instruction within the pipeline. Note that not every instruction-progress mapping constitutes a concrete pipeline state, e.g. if two different instructions are mapped to the same progress or if a fetch and data bus access are served at the same time. For the sake of simplicity, we nevertheless use the superset $\mathcal{I}_d \to P$ in the following.

To focus on the essence of progress-based abstraction, we have dropped all non-pipeline parts from the configuration space. The pipeline cycle behaviour $cycle \subseteq \mathcal{C} \times 2^{\mathcal{E}} \times \mathcal{C}$, however, still depends on the state of the

non-pipeline parts. We model the effect of the non-pipeline parts as external functions that can be queried within *cycle*, e.g. to determine whether a memory access hits the cache. As an example, consider the cycle behaviour of a five-stage in-order pipeline provided in Appendix A.1. In a more complicated system, the behaviour of the non-pipeline parts might itself depend on the pipeline behaviour, e.g. the pipeline might reorder memory accesses which influences the cache behaviour. The analysis of the non-pipeline parts that provides the valuation of the external functions has to conservatively take the possible pipeline behaviour into account. The cache analysis, for example, should consider all possible access orderings.

In the concrete domain, *cycle* is a deterministic function as the external functions provide definite answers. If abstraction is employed for the non-pipelined part, e.g. a cache abstraction, the external functions might provide uncertain answers. The cycle behaviour becomes non-deterministic because it has to account for all successor configurations permitted by the uncertain external information.

During a cycle transition, a set of events $evs \subseteq \mathcal{E}$ is emitted. An event is a descriptor for the occurrence of an action in the system, e.g. a cache miss or a newly finished instruction. An action can only happen when a program instruction progresses in the pipeline. Thus, we associate each event with the respective causal instruction instance $ins \in \mathcal{I}_d$ and a cycle transition from progress p to progress p'. Consequently, each event can happen *at most once* during a single program execution. An example of an event is

> *a cache miss caused by the second instance of the instruction at address* `0x480` *when entering the memory stage of the pipeline.*

Progress Abstraction

We assume that the progress of individual instructions in the pipeline is ordered by $\sqsubseteq_P \subseteq P \times P$ such that $p \sqsubseteq_P p'$ if p represents at least the progress p', e.g. a later or equal pipeline stage. The definition of the progress order depends on the actual pipeline construction. In Section 4.3, we provide an order on the progress of instructions within the five-stage in-order pipeline described in Appendix A.1.

This ordering can be extended to a partial order $\sqsubseteq \subseteq \mathcal{C} \times \mathcal{C}$ relating the progress of all instructions in the instruction sequence:

$$c \sqsubseteq c' \Leftrightarrow \forall ins \in \mathcal{I}_d.\ c(ins) \sqsubseteq_P c'(ins).$$

We say that configuration c has more or equal progress than configuration c' if each instruction has progressed in c at least as much as in c'.

The core of the abstraction is that an abstract configuration $\widehat{c}$ captures all concrete configurations that have made *at least* as much progress as $\widehat{c}$. We observe that it is possible to use elements from the concrete domain $\mathcal{C}$ as abstract configurations. This yields a *non-relational* abstraction since individual instructions and their progress are considered independently of each other. More powerful, relational abstractions are conceivable as we briefly sketch in Section 4.10. A detailed discussion of relational abstractions exceeds the scope of this thesis. Consequently, we choose the set of abstract configurations $\widehat{\mathcal{C}}$ as

$$\widehat{\mathcal{C}} := \mathcal{C} = \mathcal{I}_d \to P$$

with the above partial order $\sqsubseteq$ based on the notion of progress. Note, however, that not every abstract pipeline state is necessarily a valid concrete pipeline state. As an example, a state with two concurrently ongoing bus accesses can arise in the abstract but not the concrete domain.

The concretisation function $\gamma^{conf} : \widehat{\mathcal{C}} \to 2^{\mathcal{C}}$ for abstract configurations is given by

$$\gamma^{conf}(\widehat{c}) := \{c \mid c \sqsubseteq \widehat{c}\}.$$

An abstract configuration thus describes all concrete configurations with more or equal progress. The consistency of the partial order and the concretisation function (Equation 2.10) follows by definition and transitivity of the partial order.

The next step is the definition of an abstract cycle behaviour $\widehat{cycle}$ for the progress domain and the formulation of *local consistency*. Unlike the cycle behaviour of the previous non-progress-based abstractions, the cycle behaviour of a progress-based abstraction applied to $\widehat{c}$ cannot be expected to produce the same events as the cycle transition from each concrete configuration $c \in \gamma^{conf}(\widehat{c})$. The reason is that c can have more progress than $\widehat{c}$. Therefore, the concrete cycle transition exhibits events that are only produced by future abstract cycle transitions that are transitively reachable from $\widehat{c}$. This asynchronicity of event occurrences requires a generalised version of the local consistency condition.

The abstract cycle behaviour $\widehat{cycle} \subseteq \widehat{\mathcal{C}} \times 2^{\mathcal{E}} \times \widehat{\mathcal{C}}$ is *locally consistent* w.r.t. the concrete cycle behaviour $cycle \subseteq \mathcal{C} \times 2^{\mathcal{E}} \times \mathcal{C}$ if

$$\begin{aligned}
&\forall i \in I_p \, \forall \tau \in \mathcal{P}(i) \, \forall c' \in \mathcal{C} \, \forall evs \subseteq \mathcal{E} \, \forall \widehat{c} \in \widehat{\mathcal{C}}. \\
&\quad cycle(\tau_{|\tau|}.c)(evs)(c') \wedge \tau_{|\tau|}.c \in \gamma^{conf}(\widehat{c}) \\
&\qquad \Rightarrow \exists \widehat{c}' \in \widehat{\mathcal{C}} \, \exists \widehat{evs} \subseteq \mathcal{E}. \; \widehat{cycle}(\widehat{c})(\widehat{evs})(\widehat{c}') \wedge c' \in \gamma^{conf}(\widehat{c}') \wedge \\
&\qquad\qquad \bigcup_{ins \in \mathcal{I}_d} events(\tau \circ (evs, c'), ins, \widehat{c}(ins), \widehat{c}'(ins)) = \widehat{evs}. \quad (4.1)
\end{aligned}$$

The function $events(\tau, ins, p, p')$ returns the events that occur on trace τ when instruction ins progresses from p to p'. $\mathcal{P}(c)$ represents the set of partial traces that start in configuration c, i.e.:

$$\begin{aligned}
\mathcal{P}(c) := \{\tau \in \mathcal{C} \times (2^{\mathcal{E}} \times \mathcal{C})^* \mid \tau_0.c = c \wedge \\
\forall \, i \in [1, |\tau|]. \; cycle(\tau_{i-1}.c)(\tau_i.evs)(\tau_i.c)\}.
\end{aligned}$$

A trivial choice for $\widehat{cycle}$ is the identity relation, i.e. each configuration c is mapped to c again with no events happening. Clearly, this definition fulfils the local consistency criterion. However, this abstract cycle relation is useless: an initial configuration c_i is stuck without progress in the cyclic transition $\widehat{cycle}(c_i)(\emptyset)(c_i)$ and never reaches final events. Consequently, there are no traces through the abstract execution graph from initial to final configurations. Besides local consistency, we thus require $\widehat{cycle}$ to have *some* progress during each cycle transition:

$$\forall \widehat{c}, \widehat{c}' \in \widehat{\mathcal{C}} \; \forall evs \subseteq \mathcal{E}. \; \widehat{cycle}(\widehat{c})(evs)(\widehat{c}') \Rightarrow \widehat{c}' \sqsubset \widehat{c}, \quad (4.2)$$

where $\widehat{c}' \sqsubset \widehat{c}$ is defined as $\widehat{c}' \sqsubseteq \widehat{c} \wedge \widehat{c} \not\sqsubseteq \widehat{c}'$.

As the domains of abstract and concrete configurations coincide, an obvious choice for the abstract cycle transition is to reuse the concrete cycle transition relation, i.e. $\widehat{cycle} = cycle$. $\widehat{cycle}$ is guaranteed to have progress as the concrete machine is guaranteed to make progress in each step. The local consistency, leaving events aside, degenerates to a *monotonicity* property of the concrete cycle behaviour:

$$\begin{aligned}
\forall c_1, \widehat{c}_1, c_2 \in \mathcal{C}. \, c_1 \sqsubseteq \widehat{c}_1 \wedge cycle(c_1)(.)(c_2) \\
\Rightarrow \exists \widehat{c}_2 \in \mathcal{C}. \, cycle(\widehat{c}_1)(.)(\widehat{c}_2) \wedge c_2 \sqsubseteq \widehat{c}_2,
\end{aligned}$$

where (.) denotes arbitrary events. This is a *key insight*: If the concrete pipeline control behaves *monotonically* w.r.t. progress, there exists an efficient non-relational pipeline abstraction based on progress.

In general, however, modern pipelines do not behave monotonically due to features such as speculation or dynamic reordering of instructions. Pipelines that behave monotonically need to be designed carefully. In Section 4.5, we will present the *strictly in-order* pipeline that we prove monotonic. Conventional pipeline designs that do not behave monotonically require a different locally-consistent abstract cycle transition $\widehat{cycle}$. In Section 4.4 (Non-Monotonicity of In-Order Pipeline), we conclude that a relational abstraction is needed to capture the progress behaviour of such pipelines.

A non-progressing abstract cycle transition $\widehat{cycle}$ is not the only source of loops in the abstract execution graph which render the approach useless. In the progress-based abstraction, any two abstract configurations can be joined to a configuration with less or equal progress. If a configuration c' is joined with a configuration c from which c' evolved via $\widehat{cycle}$, the least upper bound is c which results in a loop in the execution graph. Using Algorithm 1 (Configurable Program Analysis) and the heuristics presented in Section 3.3, however, results in an abstract execution graph that is guaranteed to be loop-free. According to the heuristics, we only perform joins when an instruction just left the pipeline. After a join, we wait until the *next* instruction has left the pipeline before joining again. Furthermore, we limit ourselves to only join configurations that coincide w.r.t. their respective next-to-complete instruction. Combining the two arguments, we conclude that joining cannot introduce loops.

Finally, we generalise the concretisation function on abstract traces in order to account for the asynchronicity of event occurrences:

$$\gamma^{traces}(\widehat{T}) := \{\tau \mid \exists \widehat{\tau} \in \widehat{T}.\, |\tau| \leq |\widehat{\tau}| \wedge \forall i \in [0, |\widehat{\tau}|].\ \tau_i.c \in \gamma^{conf}(\widehat{\tau}_i.c) \wedge \quad (4.3)$$

$$(\forall e \in \widehat{\tau}_i.evs\ \exists k \leq i.\ e \in \tau_k.evs \vee \tau_0.c(e.ins) \sqsubset_P e.p) \wedge \quad (4.4)$$

$$(\forall e \in \tau_i.evs\ \exists k \geq i.\ e \in \widehat{\tau}_k.evs \vee \widehat{\tau}_{|\widehat{\tau}|}.c(e.ins) \sqsupseteq_P e.p)\}. \quad (4.5)$$

An abstract trace can describe shorter concrete traces. Thus we extend the notation τ_i to return the last configuration of τ and an empty set of events if i exceeds the length of τ. Furthermore, each event e is associated to an instruction instance $e.ins \in \mathcal{I}_d$ when progressing from $e.p \in P$ to $e.p' \in P$.

The concretisation function on traces uses a three-fold condition:

(4.3) As before, the i^{th} concrete configuration needs to be described by the i^{th} abstract configuration.

(4.4) An event on the abstract trace must have happened before on the concrete trace, or the concrete trace started only after the event occurred. This means, no additional events appear on the abstract trace if the concrete trace starts with an initial configuration.

(4.5) An event on the concrete trace must happen afterwards on the abstract trace, or the abstract trace stops before the event could occur. This means, no events disappear due to the abstraction if the abstract trace ends in a final configuration.

The presented progress-based microarchitectural abstraction satisfies the following correctness theorem.

Theorem 4.1.1 (Trace Coverage)**.** *Let $\widehat{\mathcal{C}}$ be a set of configurations that abstract from $\mathcal{C}$ and $\widehat{cycle}$ a relation that is locally consistent to cycle (Equation 4.1) and is guaranteed to make some progress in each cycle transition (Equation 4.2). Let G_p be an execution graph and $\widehat{G}_p$ a loop-free abstract execution graph allowing for replacements, i.e. satisfying Equations 2.5 and 2.11. The set of abstract traces* covers *all concrete traces:*

$$\mathcal{T}(G_p) \subseteq \gamma^{traces}(\mathcal{T}(\widehat{G}_p)).$$

Proof. Let a program p and a concrete trace $\tau \in \mathcal{T}(G_p)$ be given. We need to show that there exists an abstract trace $\widehat{\tau}$ through the abstract execution graph $\widehat{G}_p$ such that $\tau \in \gamma^{traces}(\widehat{\tau})$. First, we prove by induction that for each prefix trace $\tau^{(i)}$ we find an abstract partial trace $\widehat{\tau}^{(i)}$ such that $\tau^{(i)} \in \gamma^{traces}(\widehat{\tau}^{(i)})$. Second, we show that $\widehat{\tau}^{(|\tau|)}$ can be extended to $\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)$ with $\tau \in \gamma^{traces}(\widehat{\tau})$.

First Statement:
We consider $i = 0$ as induction base case. By definition $\tau^{(0)} = \tau_0$ with $\tau_0.c \in I_p$. Equation 2.5 provides an abstract initial state $\widehat{i} \in \widehat{I}_p$ such that $\tau_0.c \in \gamma^{conf}(\widehat{i})$. Choosing $\widehat{\tau}^{(0)} = \widehat{i}$ concludes the base case as no events have happened so far, i.e. $\tau_0.evs = \widehat{\tau}_0.evs = \emptyset$.

Now, we consider the induction step from i to $i + 1$. Using the induction hypothesis, we know that for $\tau^{(i)}$ there exists an abstract partial

trace $\widehat{\tau}^{(i)}$ such that $\tau^{(i)} \in \gamma^{traces}(\widehat{\tau}^{(i)})$. In particular, we can deduce that $\tau_i^{(i)}.c \in \gamma^{conf}(\widehat{\tau}_i^{(i)}.c)$. By definition of $\tau^{(i+1)} = \tau^{(i)} \circ (evs_{i+1}, c_{i+1})$, we know that $cycle(\tau_i^{(i)}.c)(evs_{i+1})(c_{i+1})$.

By using *local consistency* of the abstract cycle behaviour, instantiated with $\tau^{(i)}$ and $\widehat{\tau}_i^{(i)}.c$, we obtain

$$\exists \widehat{c}', \widehat{evs}.\ \widehat{cycle}(\widehat{\tau}_i^{(i)}.c)(\widehat{evs})(\widehat{c}') \wedge c_{i+1} \in \gamma^{conf}(\widehat{c}') \wedge \bigcup_{ins \in \mathcal{I}_d} events(\tau^{(i)} \circ (evs_{i+1}, c_{i+1}), ins, \widehat{\tau}_i^{(i)}.c(ins), \widehat{c}'(ins)) = \widehat{evs},$$

as the premises are satisfied as just shown before. This provides the abstract partial trace $\widehat{\tau}^{(i+1)} = \widehat{\tau}^{(i)} \circ (\widehat{evs}, \widehat{c}')$. It remains to be shown that $\tau^{(i+1)} \in \gamma^{traces}(\widehat{\tau}^{(i+1)})$ which comprises the three parts 4.3, 4.4, and 4.5.

The first part 4.3 follows from the induction hypothesis for all indices up to i. For the index $i+1$, the claim follows from local consistency and, if joining is employed, from the consistency of the partial order w.r.t. γ^{conf} in Equation 2.10.

For the second part 4.4, we only need to consider the newly added events $\widehat{evs}$. By local consistency, the events $\widehat{evs}$ only comprise events of $\tau^{(i+1)}$.

Part 4.5 is more involved. Let e be an event in evs_{i+1}. Depending on the progress $\widehat{c}'(e.ins)$, either the right hand side is true ($\widehat{c}'(e.ins) \sqsupseteq_P e.p$) or otherwise, by local consistency, $e \in \widehat{evs}$. This is because

$$e.p = \tau_i^{(i+1)}.c(e.ins) \sqsubseteq_P \widehat{\tau}_i^{(i+1)}.c(e.ins)$$

and

$$\widehat{\tau}_{i+1}^{(i+1)}.c(e.ins) = \widehat{c}'(e.ins) \sqsubset_P e.p,$$

i.e. $e.ins$ has progressed enough on the abstract trace to emit e as part of $\widehat{evs}$.

There is another case to account for. Consider an older event e on the concrete trace such that the condition $\widehat{\tau}_{|\widehat{\tau}|}.c(e.ins) \sqsupseteq_P e.p$ is not valid anymore. This can happen because the abstract trace has been extended by an abstract configuration with more progress during the induction step. Formally, this means that $\widehat{\tau}_i^{(i+1)}.c(e.ins) = e.p$ and $\widehat{\tau}_{i+1}^{(i+1)}.c(e.ins) \sqsubset_P e.p$. By local consistency again, we know that event e must be in $\widehat{evs}$.

This concludes the proof for the induction step, and thus the proof of the first statement.

Second Statement:
The first statement provides an abstract (partial) trace $\widehat{\tau}^{(|\tau|)}$ through the abstract execution graph $\widehat{G}_p$. We need to show that it is a complete trace, i.e. an element of $\mathcal{T}(\widehat{G}_p)$ according to Equation 2.2. After the proof of the first statement, it remains to show that the abstract trace ends with a final event $e \in \widehat{F}_p$. Since $\tau \in \mathcal{T}(G_p)$ and $\tau \in \gamma^{traces}(\widehat{\tau}^{(|\tau|)})$, the final event that terminates τ has either occurred during the last cycle of $\widehat{\tau}^{(|\tau|)}$ or has not occurred yet. In the first case, $\widehat{\tau}^{(|\tau|)}$ ends with a final event. In the second case, we extend $\widehat{\tau}^{(|\tau|)}$ to a complete trace $\widehat{\tau}$ such that $\tau \in \gamma^{traces}(\{\widehat{\tau}\})$.

The cycle behaviour of the concrete system does not end at a final configuration for program p but executes the next program. The cycle behaviour beyond p allows us to use *local consistency* to extend our abstract trace $\widehat{\tau}^{(|\tau|)}$ along the lines of the proof of the first statement. We continue until we reach a final event. We call the resulting abstract trace $\widehat{\tau}$. Note that we always reach such an event because the abstract execution graph is loop-free and the abstract cycle behaviour is guaranteed to have some progress during each cycle transition.

We need to show that $\tau \in \gamma^{traces}(\{\widehat{\tau}\})$, especially for the positions $i > |\tau|$. The conditions 4.4 and 4.5 are met by *local consistency* along the lines of the proof of the first statement. For condition 4.3, consider the configurations $\tau_i.c = \tau_{|\tau|}.c$ that are final. All instruction in $\mathcal{I}_d$ have the maximal possible progress in $\tau_i.c$. All configurations $\widehat{\tau}_i.c$ have thus less or equal progress than $\tau_{|\tau|}.c$ which satisfies the condition 4.3 in the definition of γ^{traces}.

This concludes the proof of the second statement and thus the overall correctness proof. □

As before, the above trace coverage is sufficient to conclude that weights are maximised correctly. However, there is a pitfall of counting events on abstract traces. While each event can occur at most once during a concrete execution, it can occur multiple times on an abstract trace due to the progress-based joining. Nevertheless, all occurrences of an event on an abstract trace relate to the same single event during a concrete execution. Thus, $w(\widehat{\tau})$ counts each event at most once even if it occurs multiple times.

Corollary 4.1.2 (Sound weight bound)**.** *Let the conditions of Theorem 4.1.1 be given. Furthermore, let* $w : \left(2^{\mathcal{E}}\right)^* \to W$ *be a weight function.*

The maximum of w on traces through the abstract graph provides an upper bound *on weights on concrete traces:*

$$\max_{\tau \in \mathcal{T}(G_p)} w(\tau) \leq \max_{\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)} w(\widehat{\tau}).$$

Proof. Let $\tau \in \mathcal{T}(G_p)$ be the trace with maximal weight. According to Theorem 4.1.1, there is a $\widehat{\tau} \in \mathcal{T}(\widehat{G}_p)$ such that $\tau \in \gamma^{traces}(\{\widehat{\tau}\})$. By definition of γ^{traces}, an event in τ also occurs in $\widehat{\tau}$, because $\widehat{\tau}$ ends in a final configuration and all events of the program of interest have happened: $\forall e \in \mathcal{E}.\ \widehat{\tau}_{|\widehat{\tau}|}.c(e.ins) \sqsubset_P e.p$. We conclude $w(\tau) \leq w(\widehat{\tau})$. Thus, the claim follows by the definition of max. □

For the minimisation of event-based weights, there is an analogous argument using the second part of the concretisation function and the fact that complete concrete traces start in initial configurations where no relevant event has happened yet.

4.2 Low-Level Analysis

Microarchitectural Analysis The concrete and abstract pipeline configurations presented in the previous section are based on a single sequence of unique instructions. In this section, we want to give an intuitive overview how the approach can be practically applied to whole programs. Given a program, there are usually multiple instruction sequences due to conditionals. Therefore, we consider multiple different partial sequences that span those instructions that might be in the pipeline at the same time. While executing an instruction inside a loop, multiple subsequent instances of the same instruction can occur inside the pipeline. Therefore, we symbolically distinguish multiple instances of the same instruction in an instruction sequence using indices.

As an example, consider the abstract execution graph in Figure 4.1 based on the control-flow graph from Figure 3.3. For simplicity, we illustrate a three-stage pipeline without hazards. Dashed nodes and edges correspond to nodes and edges in the control-flow-graph. Each pipeline state is associated to the instruction that finishes execution next. The end state of an instruction serves as initial state of the respective next instruction.

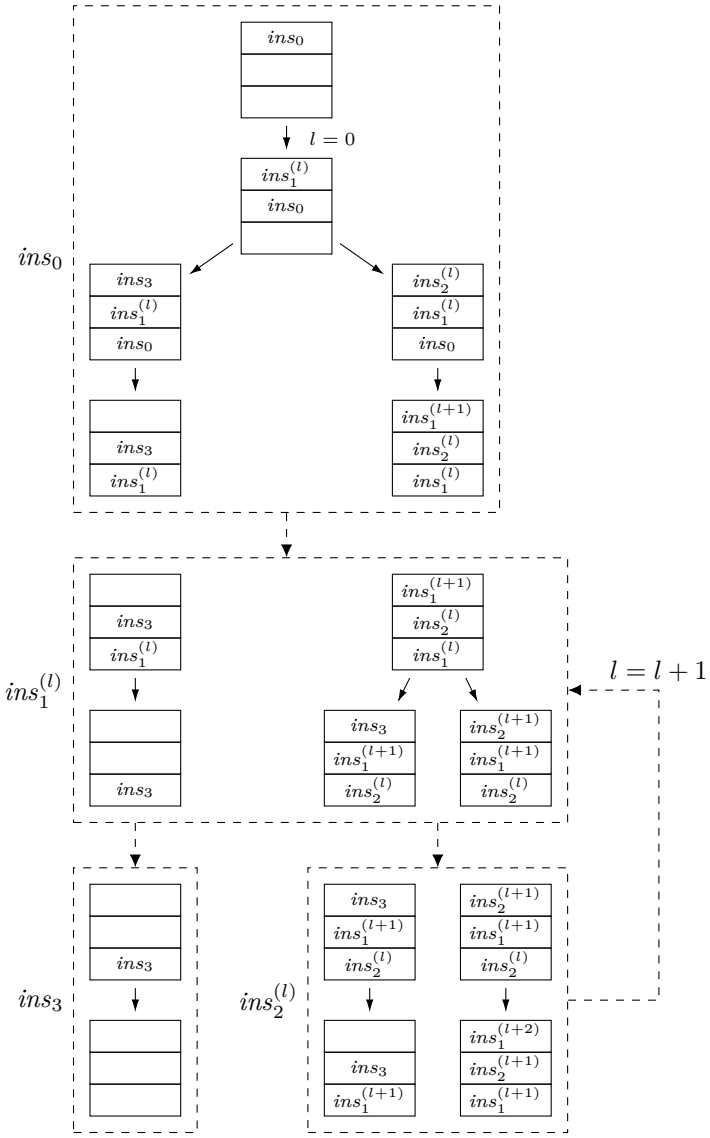


Figure 4.1: Abstract execution graph of the program given in Figure 3.3 using the progress abstraction of a three-stage pipeline.

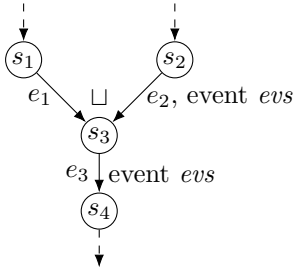


Figure 4.2: Path analysis for progress-based abstraction. Unique events can occur multiple times due to joining.

In the dashed box that corresponds to the execution of instruction ins_0, we distinguish two partial sequences of instructions depending on whether the loop is entered or not. Configurations associated to different instruction sequences are kept separate during analysis and are not considered as join candidates. They are kept separate as long as any distinctive instruction is processed in the pipeline.

The instructions within the loop are symbolically indexed by a loop iteration counter l to distinguish different instances of the same instruction. As soon as all instructions that belong to an iteration l left the pipeline, we adjust the counter ($l = l + 1$). This way, we analyse a single symbolic loop iteration that covers all behaviours of any possible loop iteration. Note that control-flow loops do not impose a problem on the correctness in Theorem 4.1.1: On each round through the loop, the configurations progress to the next loop iteration. The total number of loop iterations is bounded. This is different from the problematic loops where the progress of the pipeline is stuck.

Path Analysis Path analysis determines the maximal weight on any path through the abstract execution graph as computed by the microarchitectural analysis. For this purpose, constraints are formulated on the number of times an edge can be taken or on the number of events that can occur during execution. Although events occur at most once during a concrete execution, an event can occur multiple times on an abstract execution trace. The joining of two abstract configurations can yield a configuration that exhibits less progress than the original configurations. Consequently,

an event, which is associated to an advance in progress of an instruction instance, can occur multiple times in the abstract execution graph. As an example, see Figure 4.2. Note that this behaviour does not occur when using state-of-the-art abstractions that keep the pipeline state, and thus the progress of instructions, concrete.

In the original path analysis described in Section 3.3, we could derive the constraint $x_{e_2} + x_{e_3} \leq 1$ for the graph in Figure 4.2 since we know that the event *evs* can only occur once. However, this constraint precludes feasible abstract traces, namely the trace $s_2 e_2 s_3 e_3 s_4$, if a progress-based abstraction is used. One possible solution to express the above knowledge is to introduce a new binary variable y_{evs} to the integer linear program that describes whether the specific event *evs* occurs or not. The resulting constraints become $y_{evs} \leq 1$, $x_{e_2} \leq y_{evs}$, and $x_{e_3} \leq y_{evs}$. This technique also extends to more complex constraints such as persistence constraints.

4.3 Progress of In-Order Pipeline

We have introduced the formal background for a progress-based pipeline abstraction. Next, we instantiate the notion of *progress* of an instruction for a five-stage in-order pipeline. The progress of an instruction is determined by the pipeline stage it resides in and the number of cycles remaining to complete the current stage. We use the number of remaining cycles to model latencies of e.g. the functional units and the main memory. The set of possible progress is then defined as

$$P := \mathcal{S} \times \mathbb{N}_0$$

where $\mathcal{S}$ denotes the set of pipeline stages. The set of pipeline stages is defined as

$$\mathcal{S} := \{pre, IF, ID, EX, MEM, WB, post\}.$$

The set $\mathcal{S}$ features artificial stages to model that an instruction has not yet entered (*pre*) or has already left (*post*) the pipeline. The other elements of $\mathcal{S}$ correspond to the classic five stages: instruction fetch, instruction decode, execute, data memory access, and register write-back. In Appendix A.1, we provide a definition of the cycle behaviour of a five-stage pipeline using the above progress notation.

In static program analysis, the order of abstract elements models the notion of *precision*, i.e. $a \sqsubseteq b$ if and only if b describes at least the concrete

elements that a describes. In the progress abstraction, an (abstract) configuration describes all configurations such that instructions have advanced at least as much in the pipeline. Thus, a progress $p \in P$ is less than or equal to p' if p describes a later or equal pipeline stage. We define the order $\sqsubseteq_{\mathcal{S}}$ on pipeline stages as

$$\mathit{post} \sqsubseteq_{\mathcal{S}} \mathit{WB} \sqsubseteq_{\mathcal{S}} \mathit{MEM} \sqsubseteq_{\mathcal{S}} \mathit{EX} \sqsubseteq_{\mathcal{S}} \mathit{ID} \sqsubseteq_{\mathcal{S}} \mathit{IF} \sqsubseteq_{\mathcal{S}} \mathit{pre}.$$

Within the same pipeline stage, fewer remaining cycles indicate more progress. Thus, we define the total order of progress in a five-stage pipeline as

$$(s, n) \sqsubseteq_P (s', n') \Leftrightarrow s \sqsubset_{\mathcal{S}} s' \vee (s = s' \wedge n \leq n'),$$

where $s \sqsubset_{\mathcal{S}} s'$ is defined as $s \sqsubseteq_{\mathcal{S}} s' \wedge s' \not\sqsubseteq_{\mathcal{S}} s$. The least-upper-bound operator takes the maximum according to the above total order which corresponds to the *least* progress:

$$p \sqcup_P p' = \begin{cases} p' & : p \sqsubseteq_P p' \\ p & : \text{otherwise.} \end{cases}$$

4.4 Non-Monotonicity of In-Order Pipeline

We consider a conventional five-stage pipeline with caches and a single main memory for code and data, as given in Appendix A.1. The instruction and the data cache share a bus to the common main memory. If both caches want to start a memory access at the same time, an arbiter grants one of the caches access to the bus. Without loss of generality, we assume that accesses of the data cache are prioritised.

The cycle behaviour of the described in-order pipeline is *not monotonic*. As a counterexample, consider Figure 4.3. The problem originates from the fact that the ongoing fetch of an instruction (`add`) from main memory can block the main memory access of a previous instruction (`load`) which resides in the memory stage. In this case, more progress of the `add`-instruction turns out to be detrimental to the progress of the `load`—and potentially detrimental to the overall progress. In [Hahn et al., 2015a], we demonstrate that this non-monotonicity can result in a timing anomaly under certain circumstances.

In the context of the progress-based abstraction, non-monotonicity precludes the use of the concrete *cycle* behaviour as abstract $\widehat{\mathit{cycle}}$ behaviour

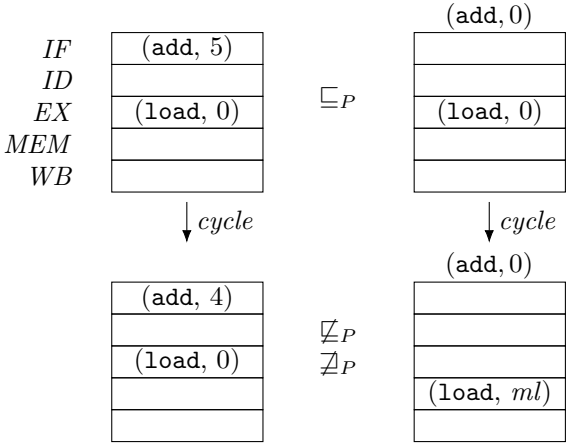


Figure 4.3: Example of non-monotonic cycle behaviour of the in-order pipeline described in Appendix A.1.

according to Equation 4.1. Can another $\widehat{cycle}$ be found that is locally consistent with *cycle*? Unfortunately, this is not possible if the progress of individual instructions is abstracted independently of each other. Consider the abstract state depicted in Figure 4.4. Assume that the fetch of the `add`-instruction and the data access of the `load`-instruction miss the cache, respectively. On the one hand, the abstract state describes a concrete state where the `load`-instruction has more progress and blocks the fetch of the `add`-instruction. On the other hand, it describes a concrete state where the `add`-instruction has more progress and blocks the data access of the `load`-instruction. Thus, no sound abstract cycle behaviour can advance `load` or `add` within the abstract pipeline state. As a consequence, any sound $\widehat{cycle}$ gets stuck for this abstract state and no upper execution-time bound can be derived.

Non-monotonicity is an issue for progress-based abstractions that operate on cycle-granularity *and* treat the progress of individual instructions separately. In Section 4.10, we briefly sketch alternative progress-based abstractions that might be able to deal with non-monotonic behaviour by changing one of these premises.

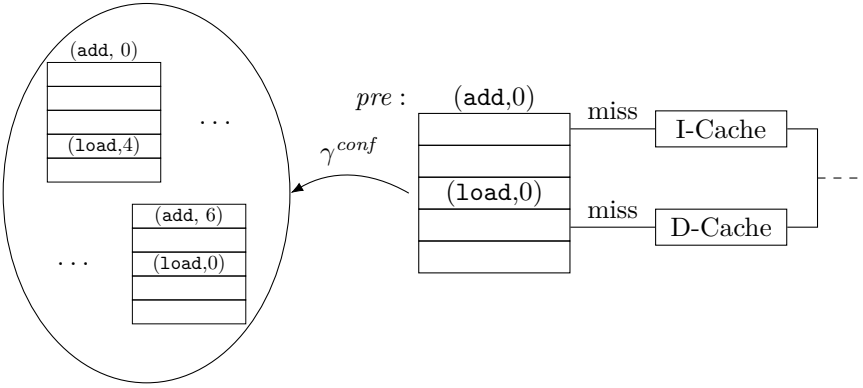


Figure 4.4: No abstract cycle behaviour can guarantee any progress when applied to this progress-based abstract state. On the left, we depict two concrete pipeline states represented by the abstract state.

In this work, we consider a memory system that shares code and data according to the von-Neumann architecture. Most modern computer systems follow this scheme because it offers more flexibility and, if used with caches, incurs only a small performance degradation due to bus contention. However, some microcontrollers feature physically separated code and data memory with individual buses and thus follow the Harvard architecture. Note that physically separate memories combined with a conventional in-order pipeline cannot trigger non-monotonic behaviour. In the next section, we show how to achieve monotonic behaviour even in the presence of shared code and data memory.

Another source of non-monotonic cycle behaviour is branch prediction with speculative instruction fetches. As described in Appendix A.1, speculative instruction fetches aim to reduce the impact of control hazards. If, however, a misprediction is encountered while a speculative fetch is performed in main memory, it is not always possible to abort the ongoing memory access. A pipeline state with more progress than another state due to having already started a speculative fetch can consequently fall behind w.r.t. progress if the prediction turns out to be wrong. This phenomenon is also known as speculation timing anomaly [Reineke et al., 2006].

4.5 Strictly In-Order Behaviour

In the previous section, we have discussed the reasons behind the non-monotonic behaviour of a conventional in-order pipeline. We want to modify the pipeline design in order to achieve a monotonic cycle behaviour while keeping as much performance of the original design as possible.

First of all, we conservatively refrain from any branch prediction and speculative memory accesses. In Section 4.9, we discuss to which extent speculation might be reintroduced in a monotonic design. Second, we ensure that all bus memory accesses, i.e. instruction fetches and data accesses, happen in program order. To this end, we delay the memory access caused by an instruction cache miss until every preceding instruction that potentially accesses the data memory reaches the memory pipeline stage. This strict bus access ordering inspired the name of the modified design: the *strictly in-order* pipeline.

Analogously to the cycle behaviour of the conventional in-order pipeline in Appendix A.1, we formally define the cycle behaviour of the strictly in-order pipeline. This definition allows us to formally prove monotonicity. As the structure of the pipeline has not changed, we use the same notion of progress as introduced in Section 4.3.

The cycle behaviour of the strictly in-order pipeline, i.e. $cycle(p)(evs)(p')$, is given by the equations in Figure 4.5. A pipeline state $p \in \mathcal{I}_d \to \mathcal{S} \times \mathbb{N}_0$ maps each instruction to its current pipeline stage and the number of remaining cycles to finish the current stage. The state can also be expressed as a pair of functions $p = (stage, cnt) \in (\mathcal{I}_d \to \mathcal{S}) \times (\mathcal{I}_d \to \mathbb{N}_0)$. $\mathcal{I}_d$ denotes the set of dynamic instruction instances from the instruction sequence processed during a specific program execution. We order the instruction instances according to their position within the sequence, i.e. $ins_n < ins_m$ denotes that instruction instance ins_n occurs before ins_m. With each dynamic instruction $ins \in \mathcal{I}_d$, we associate its operation code $opc(ins)$, its operand registers $ops(ins)$, and its destination registers $target(ins)$. Parts that do not belong to the pipeline control are modelled by external functions. The function $ichit(ins)$ ($dchit(ins)$) returns true if and only if the fetch (data) access of instruction instance ins hits the instruction (data) cache. $exlat(ins)$ returns the execution latency of instruction instance ins which might generally depend on the operand values. In our implementation, the multiply-accumulate instruction takes two cycles for execution while all other instructions execute within a single cycle. $memlat_f(ins)$ ($memlat_d(ins)$)

$$p' := \lambda i \in \mathcal{I}_d.\begin{cases}(stage'(i), latency(i)) & : ready(i) \wedge willbefree(stage'(i))\\ (stage(i), cnt'(i)) & : \text{otherwise}\end{cases}$$

$$cnt'(i) := \begin{cases}cnt(i) - 1 & : cnt(i) > 0\\ 0 & : cnt(i) = 0\end{cases}$$

$$stage'(i) := \begin{cases}post & : stage(i) = WB\\ WB & : stage(i) = MEM\\ MEM & : stage(i) = EX\\ EX & : stage(i) = ID\\ ID & : stage(i) = IF\\ IF & : stage(i) = pre\end{cases}$$

$$\begin{aligned}ready(i) := {} & cnt(i) = 0\\ & \wedge (stage(i) = ID \Rightarrow \neg ophaz(i))\\ & \wedge (stage(i) = pre \Rightarrow \neg brpending(i) \wedge\\ & \qquad next(i) \wedge\\ & \qquad (ichit(i) \vee \neg mempending(i)))\end{aligned}$$

$$\begin{aligned}willbefree(s) := {} & s = post\\ & \vee (\neg \exists i.\ stage(i) = s)\\ & \vee (\exists i.\ stage(i) = s \wedge ready(i) \wedge willbefree(stage'(i)))\end{aligned}$$

$$latency(i) := \begin{cases}memlat_f(i) & : stage'(i) = IF \wedge \neg ichit(i)\\ memlat_d(i) & : stage'(i) = MEM \wedge (\neg dchit(i) \vee opc(i) = store)\\ exlat(i) & : stage'(i) = EX\\ 0 & : \text{otherwise}\end{cases}$$

$$brpending(i) := \exists j < i.\ opc(j) = branch \wedge p(j) \sqsupset (EX, 0)$$

$$next(i) := stage(i) = pre \wedge \forall j < i.\ stage(j) \neq pre$$

$$mempending(i) := \exists j < i.\ opc(j) \in \{load, store\} \wedge p(j) \sqsupset (MEM, 0)$$

$$ophaz(i) := \exists o \in ops(i)\, \exists j < i.\ p(j) \sqsupset (MEM, 0) \wedge opc(j) = load \wedge o \in target(j)$$

$$\begin{aligned}evs := {} & \{(\text{dcmiss}, i, p(i), p'(i)) \mid p'(i) \sqsubset (EX, 0) = p(i) \wedge \neg dchit(i) \wedge opc(i) = load\}\\ & \cup \{(\text{dchit}, i, p(i), p'(i)) \mid p'(i) \sqsubset (EX, 0) = p(i) \wedge dchit(i) \wedge opc(i) = load\}\\ & \cup \{(\text{icmiss}, i, p(i), p'(i)) \mid p'(i) \sqsubset (pre, 0) = p(i) \wedge \neg ichit(i)\}\\ & \cup \{(\text{ichit}, i, p(i), p'(i)) \mid p'(i) \sqsubset (pre, 0) = p(i) \wedge ichit(i)\}\end{aligned}$$

Figure 4.5: Equations expressing the cycle behaviour of the five-stage strictly in-order pipeline. Variables i and j denote instruction instances from $\mathcal{I}_d$.

returns the memory latency that the fetch (data) access of instruction instance *ins* experiences. If abstraction is employed for the non-pipelined part, e.g. a cache abstraction, the external functions might provide uncertain answers. In this case, the cycle behaviour follows all successor configurations permitted by the uncertain external information.

From a high-level perspective, there are three possibilities how the progress of an instruction instance *ins* can evolve during a cycle transition. If the remaining cycles counter is greater than zero, the instruction instance *ins* stays in the current stage and the remaining cycles counter is decreased by one. In case the remaining cycles counter is zero, we consider two possibilities. If the instruction instance *ins* is ready and the target pipeline stage will be free from previous instruction instances, *ins* advances to the target stage. If *ins* is not ready or the target pipeline stage will still be occupied, *ins* is stalled, i.e. its progress stays unchanged.

4.6 Monotonicity

In the process of proving monotonicity of the strictly in-order pipeline, we use the following lemmas.

The definition of the cycle behaviour in Figure 4.5 uses auxiliary functions such as *ready*. We use the notation $a.f$ to denote that the auxiliary function f is evaluated for the pipeline configuration $a \in \mathcal{C}$.

Lemma 4.6.1 (Update enable)**.** *Let $a, b \in \mathcal{C}$ be two configurations. Furthermore, let $ins_i \in \mathcal{I}_d$ be an instruction with equal progress in a and b ($a(ins_i) = b(ins_i)$) and all previous instructions $ins_j < ins_i$ have progressed more in a than in b ($a(ins_j) \sqsubseteq b(ins_j)$). For any given valuation of the external functions used in ready, if b advances to the next pipeline stage, a advances as well:*

$$b.ready(ins_i) \Rightarrow a.ready(ins_i)$$
$$b.willbefree(b.stage'(ins_i)) \Rightarrow a.willbefree(b.stage'(ins_i))$$

Proof. **Part 1: *ready***
By $b.ready(ins_i)$, we get $b.cnt(ins_i) = 0$ and by $a(ins_i) = b(ins_i)$ also $a.cnt(ins_i) = 0$. For all pipeline stages except *pre* and *ID* this is sufficient to conclude $a.ready(ins_i)$. We prove the claim for *ID* and *pre* each by contradiction.

- For stage *ID*, $\neg a.ready(ins_i)$ requires an operand hazard $a.ophaz(ins_i)$. This means, there is a load instruction $ins_j < ins_i$ with progress $a(ins_j) \sqsupset (MEM, 0)$ that writes our operand. By $a(ins_j) \sqsubseteq b(ins_j)$, we conclude that $b(ins_j) \sqsupset (MEM, 0)$, i.e. there is an operand hazard in b. This contradicts $b.ready(ins_i)$.

- For stage *pre*, $\neg a.ready(ins_i)$ requires either $a.brpending(ins_i)$, or $\neg a.next(ins_i)$, or $\neg ichit(ins_i) \wedge a.mempending(ins_i)$. In all three cases, an argument analogous to *ophaz* applies. If a memory operation ins_j is pending in a during the instruction cache miss of ins_i, ins_j is also a pending memory operation in b as $a(ins_j) \sqsubseteq b(ins_j)$ during the miss of ins_i. If there is an older instruction $ins_j < ins_i$ to be fetched next, ins_j is also in the *pre* stage in b and is to be fetched next in b. If there is a branch ins_j pending in a, ins_j is also a pending branch in b as $a(ins_j) \sqsubseteq b(ins_j)$. If any of the three expressions above evaluates to true for a, it also evaluates to true for b resulting in $\neg b.ready(ins_i)$. This is a contradiction.

This concludes the proof for *ready*.

Part 2: *willbefree*
We prove the claim by contradiction. Let $s = b.stage'(ins_i)$. Assume $b.willbefree(s)$ and $\neg a.willbefree(s)$. By $\neg a.willbefree(s)$, it follows that there is an instruction $ins_j < ins_i$ in stage s in a. As $a(ins_j) \sqsubseteq b(ins_j)$, ins_j must be in stage s in b as well, because the predecessor stage is occupied by instruction $ins_i > ins_j$. As $b.willbefree(s)$ and ins_j is in s, we know that $b.ready(ins_j)$ and $b.willbefree(b.stage'(ins_j))$. By $b.ready(ins_j)$, we know that $b.cnt(ins_j) = 0$ and thus $a(ins_j) = b(ins_j)$.

We can now inductively use this Lemma 4.6.1 for ins_j which is in a later stage than ins_i. We repeat this argument until we hit either a free stage or stage *post*. Thus by applying Lemma 4.6.1 for ins_j, we get $a.ready(ins_j)$ and $a.willbefree(b.stage'(ins_j))$. This results in $a.willbefree(s)$ which contradicts our assumption. □

Lemma 4.6.2 (Progress Dependence)**.** *For the above cycle behaviour, the progress of an instruction $ins_i \in \mathcal{I}_d$ depends* solely *on the progress of previous instructions $ins_j < ins_i$ and* never *on the progress of subsequent instructions $ins_j > ins_i$.*

Proof. The progress of an instruction ins_i depends on the progress of other instructions exclusively via *ready* and *willbefree*. By Lemma 4.6.1, we know that *ready* and *willbefree* solely depend on the progress of previous instructions $ins_j < ins_i$. □

Lemma 4.6.3 (Positive Progress)**.** *The cycle behaviour applied to a configuration c yields successor configurations with more progress than c:*

$$\forall c, c' \in \mathcal{C}\, \forall evs \subseteq \mathcal{E}.\ cycle(c)(evs)(c') \Rightarrow c' \sqsubset c.$$

Proof. First, we prove $c' \sqsubseteq c$ by case distinction of *cycle*. Let an instruction $ins_i \in \mathcal{I}_d$ be given. If $c(ins_i)$ is stalled, we obtain $c'(ins_i) = c(ins_i)$ and thus $c'(ins_i) \sqsubseteq c(ins_i)$. If $c(ins_i)$ reduces the number of remaining cycles, $c'(ins_i) = (c.stage(ins_i), c.cnt(ins_i) - 1)$ and thus $c'(ins_i) \sqsubset c(ins_i)$. If $c(ins_i)$ advances to the next pipeline stage, $c'.stage(ins_i) \sqsubset c.stage(ins_i)$ and thus $c'(ins_i) \sqsubset c(ins_i)$.

To prove the strictness of $c' \sqsubset c$, it is sufficient to show that not every instruction is stalled in the pipeline. We will show that the instruction farthest down the pipeline is not stalled. Let instruction ins_i be the farthest instruction, i.e. all instructions $ins_j < ins_i$ already left the pipeline. All stages below $c.stage(ins_i)$ are empty, which results in $willbefree(c.stage'(ins_i))$. If $c.cnt(ins_i) > 0$, ins_i is not stalled as the number of remaining cycles is reduced. If $c.cnt(ins_i) = 0$, $c.ready(ins_i)$ and thus ins_i would progress to the next stage. Even if the current stage of ins_i is *ID* or *pre*, the readiness of ins_i cannot be prevented by operand hazards or pending branches/memory operations as the pipeline in front of ins_i is empty. □

Theorem 4.6.4 (Monotonicity)**.** *The cycle behaviour of the strictly in-order pipeline is monotonic.*

$$\begin{aligned}
&\forall i \in I_p\, \forall \tau \in \mathcal{P}(i)\, \forall c', \widehat{c} \in \mathcal{C}\, \forall evs \subseteq \mathcal{E}.\\
&\quad cycle(\tau_{|\tau|}.c)(evs)(c') \wedge \tau_{|\tau|}.c \sqsubseteq \widehat{c}\\
&\qquad \Rightarrow \exists \widehat{c}' \in \mathcal{C}\, \exists \widehat{evs} \subseteq \mathcal{E}.\ cycle(\widehat{c})(\widehat{evs})(\widehat{c}') \wedge c' \sqsubseteq \widehat{c}' \wedge\\
&\qquad\qquad \bigcup_{ins \in \mathcal{I}_d} events(\tau \circ (evs, c'), ins, \widehat{c}(ins), \widehat{c}'(ins)) = \widehat{evs}
\end{aligned}$$

Proof. Let $i, \tau, c', \widehat{c}, evs$ be given and let c denote $\tau_{|\tau|}.c$. Furthermore, we know that $c \sqsubseteq \widehat{c}$ and $cycle(c)(evs)(c')$. We need to prove that *cycle* applied to $\widehat{c}$ yields $\widehat{c}'$ such that $c' \sqsubseteq \widehat{c}'$ with compatible events $\widehat{evs}$.

Let $ins \in \mathcal{I}_d$ be an arbitrary dynamic instruction instance. It is sufficient to show that the concrete *cycle* behaviour relates $\widehat{c}(ins)$ with a $\widehat{c}'(ins)$ via $events(\tau \circ (evs, c'), ins, \widehat{c}(ins), \widehat{c}'(ins))$ such that $c'(ins) \sqsubseteq_P \widehat{c}'(ins)$. We distinguish three possible cases for *cycle* applied to $\widehat{c}$: (1) *ins* is stalled, (2) *ins* counts down its remaining cycles, or (3) *ins* advances to the next pipeline stage.

Pipeline stall
If instruction *ins* is stalled in configuration $\widehat{c}$, we obtain $\widehat{c}'(ins) = \widehat{c}(ins)$. By assumption, we know $c(ins) \sqsubseteq \widehat{c}(ins) = \widehat{c}'(ins)$. By Lemma 4.6.3, we conclude that $c'(ins) \sqsubseteq \widehat{c}'(ins)$. Due to stalling, no events are emitted.

Remaining cycles countdown
If instruction *ins* reduces its remaining cycles in $\widehat{c}$, we obtain $\widehat{c}.stage(ins) = \widehat{c}'.stage(ins)$ and $\widehat{c}.cnt(ins) - 1 = \widehat{c}'.cnt(ins)$. If $c(ins) = \widehat{c}(ins)$, by definition of *cycle* we obtain $c'(ins) = \widehat{c}'(ins)$. Otherwise, $c(ins) \sqsubset \widehat{c}(ins)$:

- If $c.stage(ins) \sqsubset \widehat{c}.stage(ins)$, we conclude by Lemma 4.6.3 that $c'.stage(ins) \sqsubset \widehat{c}'.stage(ins)$.
- Otherwise $c.cnt(ins) < \widehat{c}.cnt(ins)$. If $c.cnt(ins) = 0$, either *ins* advances in the pipeline resulting in $c'.stage(ins) \sqsubset \widehat{c}'.stage(ins)$ or *ins* is stalled in c resulting in $c'.cnt(ins) = c.cnt(ins) = 0 \leq \widehat{c}.cnt(ins) - 1 = \widehat{c}'.cnt(ins)$. If $c.cnt(ins) \neq 0$, by definition of *cycle* we conclude $c'.cnt(ins) = c.cnt(ins) - 1 < \widehat{c}.cnt(ins) - 1 = \widehat{c}'.cnt(ins)$.

Again, no events are emitted during the remaining cycles countdown.

Pipeline stage advance
If instruction *ins* advances in the pipeline, any possible successor $\widehat{c}'(ins)$ has stage $\widehat{c}.stage'(ins)$ and some latency. We need to find a $\widehat{c}'(ins)$ such that $c'(ins) \sqsubseteq \widehat{c}'(ins)$.

First, we show that $c'.stage(ins) \sqsubseteq \widehat{c}'.stage(ins)$ holds for any possible successor $\widehat{c}'(ins)$. We know that $c(ins) \sqsubseteq \widehat{c}(ins)$, i.e. either $c(ins) = \widehat{c}(ins)$ or $c(ins) \sqsubset \widehat{c}(ins)$. We consider the case $c(ins) = \widehat{c}(ins)$ first. As we are in the *pipeline stage advance* case, we know that $\widehat{c}.ready(ins)$ and $\widehat{c}.willbefree(\widehat{c}.stage'(ins))$. By using Lemma 4.6.1 with c, $\widehat{c}$, and *ins*, we get $c.ready(ins)$ and $c.willbefree(\widehat{c}.stage'(ins))$. Consequently, we know that instruction *ins* will also advance its pipeline stage in c which results in $c'.stage(ins) = \widehat{c}'.stage(ins)$. In the second case, i.e. $c(ins) \sqsubset \widehat{c}(ins)$, we

know $c.stage(ins) \sqsubset \widehat{c}.stage(ins)$ since we are in the *pipeline stage advance* case which implies $\widehat{c}.cnt(ins) = 0$. By definition of $stage'$, *ins* can at most move to the consecutive stage and thus $c'.stage(ins) \sqsubseteq \widehat{c}'.stage(ins)$.

As $c'.stage(ins) \sqsubseteq \widehat{c}'.stage(ins)$ and τ starts in initial configuration i, any events of *ins* for the progress transition from $\widehat{c}.stage(ins)$ to $\widehat{c}'.stage(ins)$ must have happened on $\tau \circ (evs, c')$. We consider the same valuation of the external functions *ichit*, *dchit*, *exlat*, and $memlat_{f/d}$ that has been used to generate τ. This valuation now fully determines $\widehat{c}'(ins)$. Furthermore, *cycle* emits the same events that have been seen on $\tau \circ (evs, c')$, namely $events(\tau \circ (evs, c'), ins, \widehat{c}(ins), \widehat{c}'(ins))$.

To conclude the argumentation for $c'(ins) \sqsubseteq \widehat{c}'(ins)$, we need to prove that $c'.cnt(ins) \leq \widehat{c}'.cnt(ins)$ if $c'.stage(ins) = \widehat{c}'.stage(ins)$. Immediately after the pipeline advance, $\widehat{c}'.cnt(ins)$ is the highest possible latency according to the valuation of the external functions and thus $c'.cnt(ins)$ cannot be higher because the number of remaining cycles is never increased during *cycle*. □

4.7 Anomaly Freedom

The progress-based abstraction of the strictly in-order pipeline with its monotonic *cycle* behaviour does not exhibit timing anomalies. This enables the low-level analysis to solely follow the local worst-case upon a non-deterministic choice.

All non-deterministic choices that we consider—such as cache miss/hit, (no) cache write back, (no) shared-bus blocking—have a similar impact on the pipeline state. These choices result in different latencies for the affected instructions inside the pipeline. For the rest of this section, we focus on the data cache miss/hit choice.

We prove the absence of timing anomalies related to the data cache using the sufficient criterion for domination from Theorem 2.5.4 in Section 2.5. As we are concerned about execution time, we choose $w = w_{time}$, where w_{time} yields the number of cycle transitions on a given trace. We now prove that the sufficient criterion required in Theorem 2.5.4 is fulfilled for the strictly in-order pipeline.

Lemma 4.7.1. *In the strictly in-order pipeline, the* cache miss *case dominates the* cache hit *case w.r.t. the execution time* w_{time}.

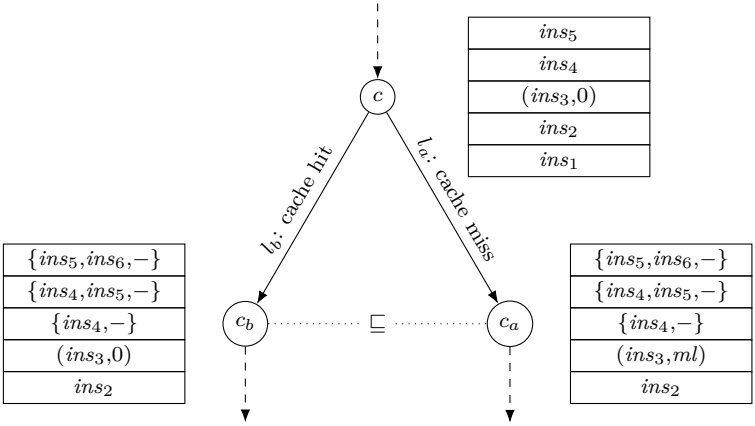


Figure 4.6: There are no timing anomalies related to the data cache hit/miss choice.

Proof. We prove the claim by using Theorem 2.5.4 and showing the required sufficient condition.

Consider Figure 4.6. In configuration c, instruction ins_3, which we assume to be a memory instruction, is about to advance to the memory stage. The time ins_3 will spend in the memory stage depends on whether the memory access performed by ins_3 hits or misses the data cache.

Consider the cache hit case resulting in c_b and ins_3 having zero cycles to wait. Depending on hazard signals which are determined by dependencies between the instructions or the instruction cache, instructions ins_4 to ins_6 could advance or stay at their current stage. We illustrate the possible positions in Figure 4.6 by using a set notation.

We pick the dominating configuration c_a such that $cycle(c)(evs_a)(c_a)$ takes the same non-deterministic choices as $cycle(c)(evs_b)(c_b)$ except that the data memory access misses the cache. Instruction ins_3 progresses to the memory stage having to wait for memory-latency many cycles to further advance. Trivially, $w_{time}(c \circ (evs_b, c_b)) = 1 \leq 1 = w_{time}(c \circ (evs_a, c_a))$. Consequently, it remains to be shown that $c_b \sqsubseteq c_a$.

The progress of instructions older than ins_3, i.e. ins_1 and ins_2, is the same in c_b and c_a as their progress does not depend on the progress of later instructions including ins_3 (Lemma 4.6.2). The progress of an instruction

$ins \in \{ins_3, ins_4, \ldots\}$ in c_b should at least be equal to the progress of ins in c_a. Whether ins advances in the pipeline depends on the predecessor configuration c and whether ins is *ready* which can depend on a non-deterministic choice. However, if *ready*(ins) holds under a data cache miss, *ready*(ins) also holds under a data cache hit. This concludes $c_b \sqsubseteq c_a$.

Note: The only dependence of *ready*(ins) on a non-deterministic choice is upon *ichit*. The absence of anomalies related to the instruction cache can still be shown analogously to the above proof. If *ready*(ins) holds under an instruction cache miss $\neg ichit$, then *ready*(ins) also holds under an instruction cache hit *ichit*. □

Using this sufficient condition in Theorem 2.5.4, we conclude that the local worst-case, the data cache miss, dominates the local best-case, the data cache hit. Thus, there are no anomalies related to data cache classifications and we can safely follow the local worst-case during low-level analysis.

4.8 Performance Evaluation

The design of the strictly in-order pipeline proposed in Section 4.5 causes more stalls than the underlying in-order pipeline. The fetch stage is stalled if a branch is pending, or if a fetch misses the cache while a data memory operation is pending in the pipeline. Consequently, the actual performance of the strictly in-order pipeline will be lower on average.

In this section, we compare the actual performance of the strictly in-order pipeline with the performance of the underlying, conventional in-order pipeline described in Appendix A. We also evaluate a non-pipelined version of the processor core for comparison. The non-pipelined core uses the same multi-cycle datapath as the in-order pipelined core, however, only one instruction is executed at a time.

In [Liu et al., 2012], Liu et al. propose a thread-interleaved five-stage pipeline with four hardware threads, called PTARM, to implement a precision timed machine designed for determinism and predictability. From the perspective of a single thread, the pipeline fetches the next instruction once the previous instruction of the same thread has finished the memory stage of the pipeline. In the meantime, the pipeline executes instructions of the three other hardware threads. The distance between consecutive instructions of the same thread is sufficient to eliminate control and data

Table 4.1: Comparison of FPGA-related characteristics of different core designs. All designs took between 2,124,544 and 2,303,488 Bits in block-RAM (caches, main memory), depending on the cache size, and 8 DSP Slices (multiplier).

Design	Max. Frequency	Logical Elements
in-order with branch prediction	62.0 MHz	5037
strictly in-order	61.3 MHz (-1.1%)	5046 (+0.2%)
non-pipelined	65.1 MHz (+5.0%)	4292 (-14.8%)
single PTARM thread	64.3 MHz (+3.7%)	4294 (-14.8%)

hazards in the pipeline which simplifies the design. In order to compare PTARM with our strictly in-order pipeline, we use a fourth variant of our core design that imitates the behaviour of a single thread in the PTARM machine. To this end, we modify the non-pipelined variant to start the next instruction once the memory stage is completed.

In the following, we first provide the characteristics of the respective FPGA implementations. Second, we determine the number of cycles needed for the actual execution of our benchmark programs.

FPGA implementation and design characteristics

All four microarchitectural design variants implement the same subset of the ARMv4 instruction set architecture. Each core design is connected via separate direct-mapped instruction and data caches of sizes between 1 KiB and 8 KiB to a unified static random-access memory of size 256 KiB, respectively. For each cache configuration, we have synthesised the four versions of our microarchitecture targeting an Altera Cyclone IV E (EP4CE115) FPGA on a Terasic DE2-115 development board. We have configured the synthesis tool to optimise for high clock frequency. The FPGA-related characteristics of the different core designs—arithmetically averaged over all cache configurations—are depicted in Table 4.1. The logical element count only covers the processor core module and excludes the logic needed to implement the memory hierarchy.

The slight increase in logical element usage for the strictly in-order pipeline originates from the circuitry needed to implement the additional

stalls. Although the additional control circuitry is not part of the longest path, any design change influences the physical layout produced by the probabilistic synthesis algorithm. Such changes in layout explain the small change in maximal clock frequency. Overall, the differences in FPGA-related characteristics between the in-order and the strictly in-order design are negligible.

The non-pipelined and the PTARM-like core improve upon the in-order pipeline in terms of clock frequency as well as logical element usage. Both changes are mainly caused by the elimination of the forwarding circuits which are not needed in these non-pipelined core designs. We discuss the impact of the increased clock frequency on the performance after the next section. In the next section, we estimate the cycle-level performance of the different core designs.

Estimation of the cycle-level performance

To assess the performance of the different core designs, we determine the respective number of clock cycles needed to execute a given program. Due to the number and size of our benchmark programs as well as the number of hardware configurations under test, an evaluation based on Verilog-level simulations has not been feasible w.r.t. time. In order to perform our experiments in real-time on an FPGA, we equip our Verilog design with a cycle counter and means to remotely control our FPGA via a USB/JTAG connection. The host processor first uploads the synthesised FPGA design, transfers the given program to the program memory on the FPGA, and resets the FPGA core to start simulation. After the program terminates, the host processor downloads the current value of the cycle counter and proceeds to the next program.

As benchmark programs, we use parts of the Mälardalen [Gustafsson et al., 2010] and TACLeBench [Falk et al., 2016] benchmark suite as well as test cases generated from model-based designs. We refer to Appendix B for more details on our benchmarks. Each benchmark program is run with a fixed input and on an initial microarchitectural state with empty caches and an empty pipeline.

For the non-pipelined, the PTARM-like, and the strictly in-order core design, we provide the cycle ratio compared to the underlying in-order pipeline, respectively. We plot the maximal and the minimal ratio as well as the geometric mean over all benchmark programs. Furthermore, we

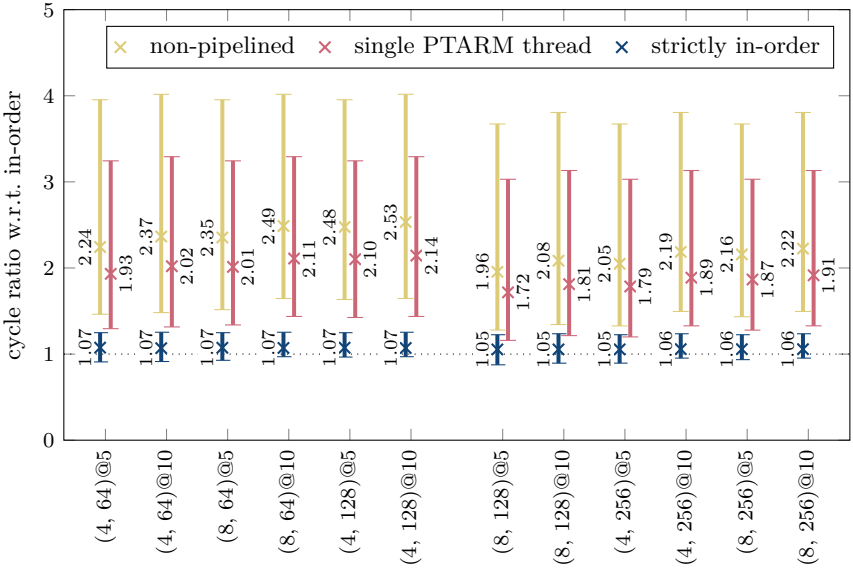


Figure 4.7: Performance in clock cycles relative to a standard in-order pipeline. Different memory configurations ((cache line size in words, number of sets)@memory word latency). Minimum, maximum, and geometric mean over all our benchmarks (*optimised* version). Lower is better.

repeat the measurements for different parameters, e.g. cache configurations and memory latencies. The results are shown in Figures 4.7 and 4.8 and discussed below.

Figure 4.7 shows the resulting cycle ratios averaged over our benchmark programs compiled with optimisations. We provide the cycle ratios for different memory configurations, i.e. different cache sizes and memory latencies. The non-pipelined design takes on average between 1.96 and 2.53 times the number of cycles needed by the in-order pipeline design (up to 4 times in the worst-case). The single PTARM hardware thread improves on the non-pipelined design by roughly 14% as an instruction fetch starts as soon as the previous instruction completes the memory phase. Thus, a single thread of the PTARM core takes on average between 1.72 and

2.14 times the number of cycles needed by a conventional in-order pipeline. The reduced single-thread performance has motivated the design of a more flexible thread-interleaved pipeline which allows multiple instructions of the same thread to execute in a pipelined fashion [Zimmer et al., 2014].

The high variance in cycle-level performance of the non-pipelined designs compared to the in-order pipeline is explained by the different demands of the benchmark programs. A memory-intensive program that causes many cache misses does not profit much from the in-order pipeline since the cache misses stall the execution. In contrast, a program whose accesses mostly hit the cache during execution experiences a significant speedup due to the better utilisation of the pipeline.

The strictly in-order pipeline needs on average 5% to 7% more clock cycles than the underlying in-order pipeline. This originates from the missing branch prediction and the additional stalls of instruction cache misses when a data memory operation is pending. Overall, the strictly in-order pipeline preserves most of the benefits of pipelined execution.

Figure 4.7 also illustrates the influence of different memory configurations (i.e. cache size and memory word latency) on the relative performance of the designs. On average over our benchmarks, larger caches increase the relative performance of the conventional in-order pipeline compared to the non-pipelined and PTARM-like design. The lower cache miss rates reduce the impact of waiting for main memory on the performance and thus emphasise the difference between pipelined and non-pipelined execution. Larger caches have almost no impact on the relative performance of the strictly in-order pipeline. On the one hand, the overhead of enforcing the strict access order on the bus diminishes with lower instruction cache miss rates. On the other hand, a lower data cache miss rate does not reduce the overhead introduced by enforcing the strict order of memory accesses on the bus: A pending memory-accessing instruction blocks an instruction cache miss even if it finally turns out to hit the data cache. A lower data cache miss rate reduces the *absolute* number of clock cycles required by the strictly and the conventional in-order pipeline by roughly the same amount. Thus, the *relative* performance of the strictly in-order pipeline can even decrease slightly for a lower data cache miss rate.

As opposed to the effect of a larger cache, higher memory latencies increase the influence of waiting for main memory on the overall performance. Thus, performance differences between the microarchitectural core designs tend to vanish for higher memory latencies. Consequently, the relative performance

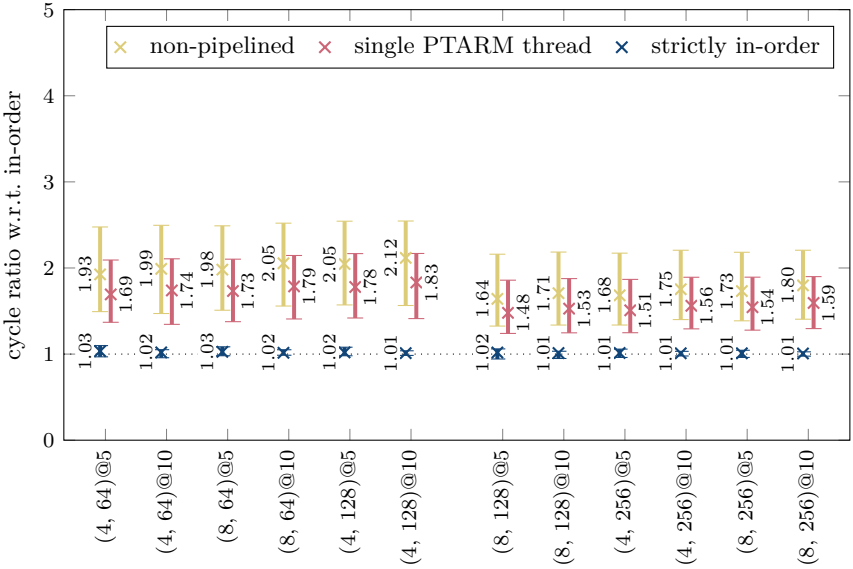


Figure 4.8: Performance in clock cycles relative to a standard in-order pipeline. Different memory configurations ((cache line size in words, number of sets)@memory word latency). Minimum, maximum, and geometric mean over all our benchmarks (*non-optimised* version). Lower is better.

of all three designs w.r.t. the conventional in-order pipeline improves. While the strictly in-order pipeline needs on average 7% more cycles to execute a program for a memory word latency of 5 cycles, the overhead reduces to 5% for a memory word latency of 10 cycles.

As it has been and probably still is common to employ non-optimised programs in safety-critical embedded systems [França et al., 2011], Figure 4.8 shows the cycle ratios averaged over our benchmarks compiled without optimisations. Non-optimised programs have a higher density of memory operations within their binaries compared to optimised programs. For our set of benchmarks, we identified 50% of all instructions in the non-optimised binaries as memory operations—in contrast to 28% for optimised binaries. Consequently, the influence of the memory hierarchy on the overall

performance is significantly higher. The higher cache miss rates due to the larger memory footprint of the programs as well as the higher number of store instructions result in more accesses to main memory. The dominant share of a program's execution time is thus spent waiting for main memory and not for the potentially pipelined execution of instructions. Hence, this results in the lower cycle ratios shown in Figure 4.8.

Towards a fair comparison with PTARM

In the previous section, we have compared the relative performance of our different microarchitectural designs using the number of needed clock cycles for execution. As an example, consider the memory configuration with a word latency of 10 cycles and caches with 128 sets and 8 words per line. The results given in Figure 4.7 show that a single thread of the PTARM core takes—averaged over all programs compiled with optimisations—1.78 times more cycles than our strictly in-order pipeline.

A fair comparison between (a multi-core variant of) our strictly in-order system and the actual PTARM system [Liu et al., 2012] has to consider more than the single-thread performance in clock cycles. Although a thorough evaluation is out of the scope of this thesis, we list important aspects that need to be considered in a fair comparison. In addition, we derive rough estimates on the relative performance accounting for the listed aspects.

Perceived Memory Latency While the PTARM core can run at a higher clock frequency than the strictly in-order core, the main memory's clock frequency cannot be increased (arbitrarily). In terms of processor cycles, a memory access perceives a "higher" latency when executed on a core that is clocked at a higher frequency. Furthermore, due to the nature of a thread-interleaved pipeline with four threads, a single thread can use the memory stage only in every fourth cycle. As a consequence, the perceived latency in cycles is a multiple of four. In our example scenario, the additional latency to load a whole cache line amounts to 17 cycles—10 cycles to load the first word plus one cycle per additional word of the cache line. Accounting for the increased clock frequency of the PTARM core and rounding to the next multiple of four yields a perceived overall latency of 20 cycles. The average ratio of processor cycles needed for execution on the PTARM core compared to the strictly in-order core then becomes 1.91.

Clock Frequency In Table 4.1, we show the maximal clock frequency for each design as computed by the synthesis tool. In the end, we are interested in the ratio of the actual execution time rather than the cycle ratio. Thus, we need to account for the difference in maximal clock speed between the strictly in-order and the PTARM-like design. The difference sums up to 4.8%. The ratio of execution time can now be obtained from the cycle ratio: 1.82.

Memory Hierarchy The PTARM system as introduced in [Liu et al., 2012] employs a fast scratchpad memory instead of caches and it accesses the DRAM via a specifically designed controller that allows the interference-free access for the four hardware threads. As noted in [Liu et al., 2012], it is challenging to efficiently program the PTARM system with its specific memory hierarchy. In contrast, our strictly in-order processor is equipped with a standard hierarchy with separate instruction and data caches and a background memory that serves one access at a time. In order to circumvent the complexity of comparing the different memory hierarchies, our PTARM-like design which focuses on the single-thread perspective uses the same memory hierarchy as our strictly in-order design. As a consequence, the single thread could fully use instruction and data caches which serve a similar purpose as the scratchpad in the original design. In a system with four threads, however, the threads would need to share the space of a local, fast memory. To account for this spatial sharing, we evaluate the relative performance of the single PTARM thread when reducing the cache size from 4KiB to 1KiB. The impact of the spatial sharing on the execution time heavily depends on the actual benchmarks. For a memory word latency of 10 processor cycles, the number of needed cycles increased by 16% on average over our benchmarks. Taking this increase into account, the ratio of execution time becomes 2.11.

Implementation Cost The implementations of the different designs take a varying amount of resources, e.g. die space for a hardware implementation, or number of logical elements on an FPGA. Thus, a varying number of cores can be fitted on a chip of a given fixed size. Our PTARM-like design with a single thread needs 4294 logical elements. To support four threads, it requires three additional sets of the 16 general-purpose 32-bit registers, i.e. 1536 logical elements, which leads to a total demand of 5830 logical

elements. Compared to our strictly in-order design (5046 logical elements), a four-thread PTARM design needs at least 15.5% more elements on an FPGA. Accounting for the difference in implementation cost, the ratio of execution time per logical element becomes 2.44.

Multiple Threads So far, we have focused on the relative performance of a single thread. Executing four independent threads in the thread-interleaved pipeline of the PTARM core takes no longer than executing only the longest thread in isolation. In the strictly in-order pipeline, however, the threads need to be executed sequentially and thus their execution times add up. In the worst case, all four threads have the same execution time demand. Compared to the execution of a single thread, the execution time of the PTARM core remains unchanged while the strictly in-order pipeline takes four times longer than before. Thus, the ratio of overall execution time per logical element reduces to 0.61. In terms of throughput, the thread-interleaved PTARM can execute 1.64 times as many instructions per time unit and per logical element as the strictly in-order pipeline.

Summary

To summarise, our above evaluation has demonstrated that the performance penalty for enforcing a strict access order in an in-order pipeline amounts to around 6%. In exchange, we are able to prove anomaly freedom and compositional timing behaviour for the strictly in-order pipeline, which helps to simplify timing analysis. The impact on analysis time is explored later in Section 5.6.3 (Compositional Base Bound) in the context of the compositional analysis of shared-bus interference.

In addition, we have presented our findings concerning a fair comparison between the strictly in-order processor and the thread-interleaved PTARM. The designs differ roughly by a factor of two w.r.t. single-thread performance—in favour of the strictly in-order processor—and instruction throughput—in favour of the thread-interleaved PTARM.

4.9 Outlook: Monotonic Extensions

In this section, we take a closer look at a selection of microarchitectural features used in the past to increase the performance of processors. Unfor-

tunately, these features have been designed without predictability in mind. Starting from the above strictly in-order pipeline, we want to briefly discuss whether and to which extent these features could be incorporated in our design without sacrificing the monotonicity property. Note that this section is rather speculative by nature because a formally thorough treatment is outside the scope of this thesis.

Store Buffer

A store buffer allows the pipeline to advance its execution without waiting for a store to complete. We describe the functionality of store buffers including their design parameters in Appendix A.2.4.

In this section, we briefly discuss whether a strictly in-order pipeline with added store buffer still behaves monotonically. There is no simple answer to this question as the answer depends on the design choices made for the store buffer. In particular, the policy when to flush the buffer plays an important role.

First, we consider a store buffer without forwarding and coalescing that drains a single entry in FIFO order upon a store request when the buffer is full. We extend the set of pipeline stages by two additional stages STB and ST, which represent a store instruction being present in the store buffer or being executed in memory, respectively. While store instructions now pass through the separate STB and ST stages until completion, non-store instructions still pass through the WB stage. Note that, in contrast to the other pipeline stages $WB \ldots IF$, multiple different instructions can be present in the STB stage according to the size of the store buffer. The order $\sqsubseteq_{\mathcal{S}}$ on pipeline stages could be refined as

$$post \begin{array}{c} \sqsubseteq_{\mathcal{S}} \\ \sqsubseteq_{\mathcal{S}} \end{array} \begin{array}{c} WB \\ ST \sqsubseteq_{\mathcal{S}} STB \end{array} \begin{array}{c} \sqsubseteq_{\mathcal{S}} \\ \sqsubseteq_{\mathcal{S}} \end{array} MEM \sqsubseteq_{\mathcal{S}} EX \sqsubseteq_{\mathcal{S}} ID \sqsubseteq_{\mathcal{S}} IF \sqsubseteq_{\mathcal{S}} pre.$$

Based on this notion of progress and the above flush policy, we conjecture that the overall system still behaves monotonically. Consider two configurations $c \sqsubseteq c'$. Assume a store buffer entry needs to be drained due to an advance in progress of an instruction *ins* during the next *cycle* starting from c, e.g. *ins* performing a store to a full buffer or a load that collides with a store buffer entry. Starting from c', the same store buffer entry needs

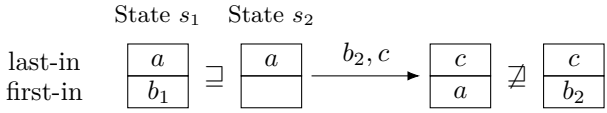


Figure 4.9: Coalescing b_1, b_2 causes non-monotonic behaviour for access sequence b_2, c on the given store buffer states. s_2 has progressed further than s_1 ($s_2 \sqsubseteq s_1$) since the instruction that stored b_1 has already finished execution in s_2.

to be drained during a future *cycle* transition—due to $c \sqsubseteq c'$—as soon as instruction *ins* reaches the required level of progress.

Forwarding can render the overall system non-monotonic: a store buffer entry in c' which is already written back in $c \sqsubseteq c'$ could be used to forward the data requested by a load. In configuration c, the requested data is instead fetched from the main memory which takes longer.

Coalescing makes the store buffer behave as an ordinary FIFO cache which is known to exhibit timing anomalies [Berg, 2006]. A coalescing store buffer behaves non-monotonically w.r.t. the above order $\sqsubseteq_{\mathcal{S}}$ as illustrated by the example in Figure 4.9. Coalescing store buffers that employ the least-recently-used retirement order are not prone to such effects and thus might keep the overall system monotonic.

For the above store buffer designs, the sequence of memory accesses is sufficient to determine their behaviour. In case of the strictly in-order pipeline, this sequence does not depend on the pipeline state and can be derived at the instruction-set-architecture level. Thus, similar to caches, the behaviour of store buffers, even with forwarding and coalescing, can be analysed prior to the pipeline analysis using techniques from cache analysis. The pipeline analysis then uses the results of the store buffer analysis to determine possible latencies of load and store instructions.

Last, we examine a store buffer that can flush entries that persist for too long—independent of the pipeline progress. Consequently, the progress of a store buffer entry depends on the time spent waiting in the store buffer. The longer a store has waited in the buffer, the more progress the associated store instruction has.

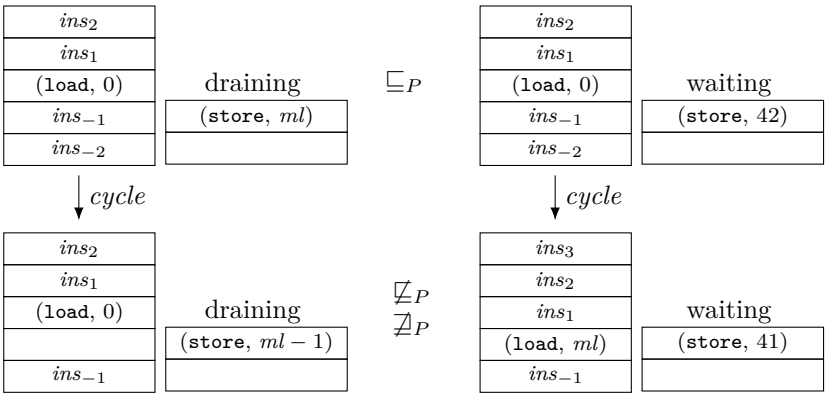


Figure 4.10: Non-monotonic cycle behaviour of a two-entry store buffer with a time-based draining strategy. On the left, the store buffer drains an entry to main memory. On the right, the store buffer still waits to reach the time threshold.

If a certain time threshold is reached, the respective store buffer entry is drained—regardless of the pipeline state. Figure 4.10 shows an example for the non-monotonic behaviour of such a system. Initially, the left configuration has more progress than the right one: the pipelines' progress is identical and the store buffer on the left already drains an entry while the store buffer on the right is still waiting. The pipelines are about to start a load memory access in the next cycle. As the store buffer on the left drains an entry, the load memory access has to wait for the bus to become free again. The pipeline on the right starts the load memory access immediately and drains the store buffer later—potentially without any conflict on the bus. For systems with a fully-pipelined memory path, the above problem might not occur as the bus is never blocked by previous accesses.

To summarise, whether or not a strictly in-order pipeline equipped with a store buffer behaves monotonically depends on the specific design decisions. On the one hand, we sketched that it is possible to use store buffers in monotonic systems. On the other hand, we demonstrated by example that certain store buffer features found in realistic systems break monotonicity.

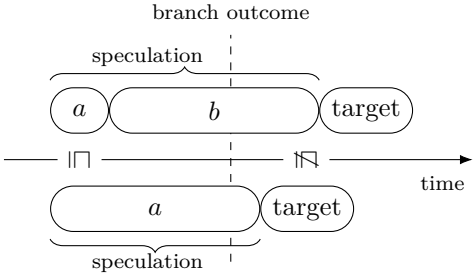


Figure 4.11: Branch speculation can result in non-monotonic behaviour in the fetch stage as memory accesses cannot be preempted. This can lead to speculation anomalies as shown above.

Speculative Execution

The strictly in-order pipeline we have proposed above does not exhibit speculative processing of instructions. In particular, upon a branch whose condition evaluates late, the pipeline stalls the instruction fetch instead of speculatively fetching from the predicted branch target.

While speculation can increase the performance, it generally leads to non-monotonic behaviour of the system. The non-monotonicity originates from the fact that the pipeline might not be able to immediately stop all speculative actions once a misspeculation is detected. For example, a (speculative) memory access can usually not be aborted immediately but only when the memory subsystem is ready.

In Figure 4.11, we illustrate an example for non-monotonic behaviour which is known as speculation anomaly [Reineke et al., 2006]. After the fetch of a branch instruction, we speculatively continue fetching from address a. In case of a fast cache hit (more progress), the speculative fetching advances to address b which then misses the cache. As soon as the misspeculation is detected, the access to b cannot be aborted but must be finished first. The case of a missing the cache thus turns out to be the overall faster case.

If speculative actions can be undone immediately once a misspeculation is detected, we conjecture that speculation can be employed in a monotonic pipeline. The behaviour of the memory subsystem determines whether an ongoing memory access can be stopped at no additional cost. This can

be possible in specific scenarios, e.g. a fully-pipelined datapath to a static memory with single-cycle latency [de Dinechin et al., 2014]. However, in general, this is not the case: in DRAM-based systems for example, there is a penalty related to switching the active row.

Even if speculative memory accesses are prohibitive, the pipeline could nevertheless speculatively fetch instructions from the cache as long as the cache is hit. Instruction cache hits are usually served within a single cycle. Thus, when a misspeculation is detected, the pipeline can flush the speculatively fetched instructions and continue normal execution without any additional delay. Upon a cache miss, the pipeline would still stall the fetch stage until the branch condition is evaluated.

However, speculation, even speculation-while-hit, has an impact on the cache behaviour: the access sequence including speculative accesses now depends on the pipeline states during execution. Under speculation-while-hit, the behaviour of caches that are not sensitive to cache hits, such as direct-mapped or set-associative caches with FIFO replacement, can still be analysed at the ISA level. The analysis of caches with LRU replacement would need to conservatively approximate the effects of potential, speculative accesses. Another, hardware-based, approach to catch the effects of speculative cache hits is to employ what one might name a *timing-committed* cache state. There is a shadow copy of the replacement state that is updated upon speculative accesses while the original replacement state stays unchanged. If the speculation turns out to be correct, the shadow state is copied into the original state; otherwise, the shadow state is reset to the original state. When using this mechanism, speculative cache hits do not influence the replacement policy's logical state and thus future cache behaviour. The term "timing-committed" is inspired by the term "commit" used to describe that a (speculatively) executed instruction takes effect on the logical machine state [Smith and Pleszkun, 1985].

Interrupts

Interrupts are a mechanism to enable the pipeline to react to external events irrespective of the ongoing execution. Upon an interrupt, the current execution is paused, the external event is handled using a special service routine, and finally the original execution is resumed. Interrupts are used for example to implement non-cooperative, preemptive scheduling using a

timer that generates interrupts periodically. Similar mechanisms are used to implement exceptions (e.g. division by zero) or privileged system calls.

The straightforward realisation of the interrupt mechanism likely results in non-monotonic behaviour. If an interrupt is detected in a certain pipeline stage, the instructions in later stages complete normally while instructions in earlier stages are aborted. Similar to the speculation example, a pipeline state with more progress that just started an instruction fetch memory access can turn out detrimental for the overall progress in case an interrupt occurs.

The non-monotonicity described above can be avoided at the expense of a longer interrupt latency, i.e. the latency between detecting and servicing an interrupt. As an example, an artificial delay between detecting and servicing could be introduced such that the pipeline is guaranteed to complete all potentially ongoing memory accesses during this delay. The additional delay is incurred in any case, even if no memory accesses are ongoing. This prolongs the average-case interrupt latency. The worst-case interrupt latency remains the same because a longer artificial delay brings no further advantage.

However, the non-monotonic behaviour upon interrupts is not a real issue. Interrupts are calls to the underlying execution environment such as an operating system that the high-level system analysis takes care of. Thus, we want to decompose the behaviour of interrupts from the uninterrupted execution of a program. In Section 5.3 (Compositionality by Hardware Design), we show that interrupts can be treated in a compositional manner as long as the cycle behaviour under the absence of interrupts is monotonic. Consequently, low-level analysis only needs to consider uninterrupted executions of a program.

Out-of-Order Execution

The execution of instructions in program order limits the amount of instruction-level parallelism that can be exploited to increase performance. Dynamic scheduling algorithms implemented in hardware, e.g. Tomasulo's algorithm [Tomasulo, 1967], can execute instructions as soon as their operands are ready. This can lead to instructions being executed out-of-(program)-order. Furthermore, pipelines featuring out-of-order execution have multiple functional units that operate in parallel. For each functional unit, the dynamic scheduling algorithm greedily selects an instruction for execution

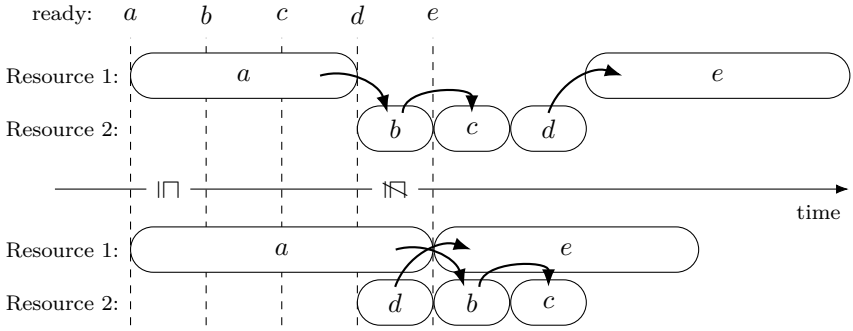


Figure 4.12: Scheduling anomaly. A shorter latency of a entails the overall worst case. Arrows denote precedence constraints. Instructions a and e execute on resource 1 while, b, c, and d execute on resource 2.

among all instructions with ready operands. As an example, the scheduler might select the oldest instruction that is ready.

Multiprocessing systems are generally known to exhibit timing anomalies for a long time [Graham, 1969]. Consequently, out-of-order execution is prone to timing anomalies as well [Lundqvist and Stenström, 1999]—due to its non-monotonic behaviour. Consider the example in Figure 4.12 that has been presented in [Li et al., 2006]. A longer execution latency of instruction a results in a reordering of instructions d, b, and c which suits the further execution better. Thus the longer latency case leads to a shorter overall execution time.

Such instruction reorderings that can cause non-monotonic behaviour are essential for dynamic scheduling. Modifying an out-of-order pipeline to support monotonicity—if possible at all—will likely come at a high price in terms of additional hardware cost and/or reduced performance.

Over-provisioning of functional units might be an option to avoid the above behaviour. Imagine the system features another instance of resource 2. In this case, instruction d could always be executed as soon as it is ready and e would finish at the same time independent of instruction a. The required headroom of additional resources should generally be related to the number of instructions that can be overtaken, i.e. the size of the reorder window. Replicating resources will be expensive and small reorder windows will limit the possible reorderings and thus the performance. Furthermore,

it is an open question whether the above mechanism is really sufficient to provide monotonicity.

Another solution might be to allow reorderings only within individual functional units. While the computations of instructions can be performed out-of-order, their results are queued at the end of the functional unit and put onto the common data bus in program order. In the case below the time axis in Figure 4.12, d could be computed out-of-order while its result is made available right after c. Consequently, the execution of e is delayed and does not finish earlier than in the case above the time axis. The technique is only useful if the execution latency is (significantly) larger than the latency to put the results on the data bus. Furthermore, the functional units must be preemptable to avoid that an instruction is blocked by the ongoing execution of younger instructions. The performance gain over a strictly in-order pipeline is, however, unclear. Again, a formal proof of monotonicity for such a pipeline is required.

To summarise, out-of-order execution is hard to turn monotonic. Even if it might be possible, the gain in performance might not be worth the effort. Nevertheless, the sketched ideas could be explored as future work.

4.10 Outlook: Enriched Abstractions

In Section 4.4 (Non-Monotonicity of In-Order Pipeline), we discussed that non-monotonicity is an issue for progress-based abstractions that operate on cycle-granularity *and* treat the progress of individual instructions separately. In this section, we briefly sketch two similar alternative progress-based abstractions that can deal with non-monotonic behaviour.

The first approach still treats the progress of individual instructions separately, but operates at coarser granularity than processor cycles. Recall the example in Figure 4.4 that depicts an abstract configuration $\widehat{c}$ where two instructions compete for the memory. In the described set of concrete configurations $\gamma^{conf}(\widehat{c})$, there are configurations that perform the instruction fetch or the data memory access in the next cycle, respectively. Thus, for neither of the instructions, progress can be guaranteed in the *next* cycle. What information can be derived if we consider a longer time span at once, e.g. twice the worst-case memory latency? For each concrete configuration in $\gamma^{conf}(\widehat{c})$, both the instruction fetch and the data memory access will have progressed to their respective next stage after that many cycles. Either

the fetch has been followed by the data access or vice versa. Thus, it is possible to derive a $\widehat{cycle}$-relation that progresses in each step and is locally consistent w.r.t. the *iterated cycle*-relation. In [Li et al., 2006], Li et al. approximate the maximal time interval for which an instruction can occupy a stage of an out-of-order pipeline. Their analysis operates at a granularity coarser than individual processor cycles which circumvents issues caused by non-monotonic behaviour.

Besides the progress of individual instructions, a *relational* analysis approach additionally approximates the combined progress of pairs of instructions. Although we do not know which of the two instructions progresses in the next cycle, we know that *at least one* of the two instructions progresses. At some point, the combined progress exceeds twice the memory latency. Similar to the first approach, we can then *normalise* our abstract configuration by advancing both instructions to their next respective pipeline stage. We can use our usual criterion for local consistency but we need to additionally prove the correctness of the normalisation operator.

In the end, both approaches are based on the same insight: global knowledge can help to provide progress-based abstractions for non-monotonic systems. However, the derivation of such global knowledge is hard, especially for more complex systems. Furthermore, coarser abstractions will introduce additional pessimism to the analysis.

Chapter 5

Achieving Timing Compositionality

Compositionality is the foundation of the separation of the timing verification problem into the low-level analyses and the higher-level schedulability analyses. The separation of concerns enables a more efficient overall timing analysis. Furthermore, different research communities can focus on their respective subproblem: low-level or schedulability analysis. The validity of the compositionality assumption is usually taken for granted without a closer look at the system under analysis.

In this chapter, we shed some light on this often neglected compositionality assumption. First, we recap the meaning of compositionality from the perspective of an actual analysis. Second, we show by experiments that even rather simple systems do not behave compositionally. We provide explanations why this is the case. Last but not least, we discuss three approaches to achieve compositionality and provide the respective proofs.

5.1 Validation of Compositionality Assumption

Recall the example scheduling interface from Section 3.5 (Schedulability Analysis). An upper bound R_i on the response times of a task i is composed from constituents provided by low-level analyses as follows:

$$R_i = C_i + Core_i(R_i) + Cache_i(R_i) \cdot ml + Bus_i(R_i) \cdot ml + Dram_i(R_i) \cdot rl,$$

where C_i denotes the non-interfered execution time, $Core_i$, $Cache_i$, Bus_i, $Dram_i$ the interference on the respective shared resource, and ml (rl) the access (refresh) latency of the main memory. Note that most approaches to schedulability analysis in literature—especially in multi-core timing analysis—use similar formulas to calculate bounds on the response times:

Atanassov and Puschner [2001]; Schliecker and Ernst [2010]; Schranzhofer et al. [2011]; Altmeyer et al. [2015].

Let I_j denote the amount of interference on the j-th shared resource, e.g. the shared bus, with the respective penalty factor p_j, e.g. the memory latency. We can rewrite the formula above as a function in terms of I_j:

$$R_i(I_1, \dots I_n) = C_i + \sum_{j=1}^{n} I_j \cdot p_j. \tag{5.1}$$

In other words, the response time bound R_i is assumed to behave *linearly* w.r.t. the amount of interference.

In 2015, Jacobs et al. propose a low-level analysis that computes the worst-case response time of a non-preemptive task given a certain amount of interference on a shared bus with round-robin arbitration. The low-level analysis can capture *fine-grained effects* on the task's timing caused by the given amount of shared-bus interference. In order to validate the linearity of R_i in terms of I_j, we use the approach proposed by Jacobs et al. to sample upper bounds on the response times of a task for several values of I_j. We call the resulting curve the *interference response curve* of the given task.

As an example, we consider the non-preemptive execution of tasks on a dual-core machine with round-robin event-driven shared bus arbitration. The cores feature a five-stage in-order pipeline with private caches and a single-entry store buffer. Figure 5.1 shows the sampled interference response curves for selected programs of our benchmark pool. We use 40 samples per program to obtain the respective curve.

The x-axis corresponds to the only source of interference: shared-bus blocking. Due to round-robin arbitration, each memory access can be blocked by at most one concurrent access per co-running core. Thus, there is a well-defined maximal amount of interference per program on a dual-core machine: *MD*, the maximal number of memory accesses performed by the program. The y-axis shows the additional execution time due to interference, relative to the direct effect of the maximal interference $MD \cdot ml$. The curve of a program behaving compositionally would have a slope of one with a y-intercept of zero.

We make the following observations:

Observation 5.1.1. *For some programs, e.g. statemate, the additional execution time due to shared-bus interference exceeds the expected penalty.*

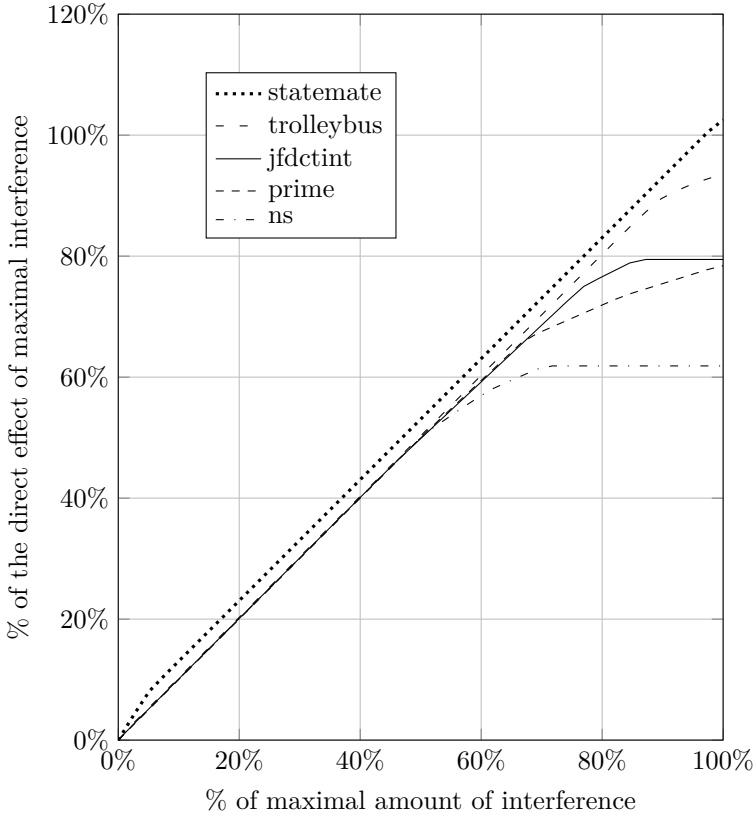


Figure 5.1: Interference response curves for selected programs.

Observation 5.1.2. *For some programs, e.g. prime, a significant portion of the expected additional latency is hidden.*

Observation 5.1.3. *For some programs, e.g. ns, additional interference beyond a certain amount does not increase the execution time further.*

In the following, we provide detailed explanations for these observations.

Amplifying Timing Anomalies A non-deterministic choice during low-level analysis where the local worst case does *not* imply the *global* worst case is commonly known as a timing anomaly (Section 2.5). The implications of timing anomalies on the efficiency of low-level analysis are discussed in the literature, e.g. in [Reineke et al., 2006]. Such anomalies are in general not an issue for the validity of the compositionality assumption.

However, Lundqvist and Stenström [1999] already described a second type of anomaly. A prolongation in the local worst case, e.g. an increase in execution latency of a single instruction, can lead to a global increase in execution time that *exceeds* the local prolongation. While servicing the local worst case, the abstract pipeline state can change in a way that leads to an *indirect effect*, i.e. an additional execution time increase beyond the local prolongation. Accordingly, we term the local prolongation the *direct effect*. We call a non-deterministic choice an *amplifying anomaly* if the overall execution time increases by more than the direct effect, i.e. there is an indirect effect.

In the above scenario, an amplifying timing anomaly occurs if a concurrent memory access, which occupies the shared bus for *ml* cycles, leads to an increase in worst-case response time of more than *ml*. Obviously, amplifying timing anomalies render the compositional response time calculation *unsound*. Ignoring indirect effects leads to estimates that do not constitute upper bounds, which is supported by our Observation 5.1.1.

Amplifying anomalies are known to occur in the analysis of complex, dynamically scheduled processors. We observe amplifying timing anomalies even in the analysis of low-complexity processors comparable to commercial microcontrollers. In the following, we provide an example that we encountered during the low-level analysis of the benchmark program `bsort100.c` taken from the Mälardalen benchmark suite [Gustafsson et al., 2010]. We found this anomaly during the analysis of a conventional five-stage in-order pipeline with a single-entry store buffer as described in Appendix A. As

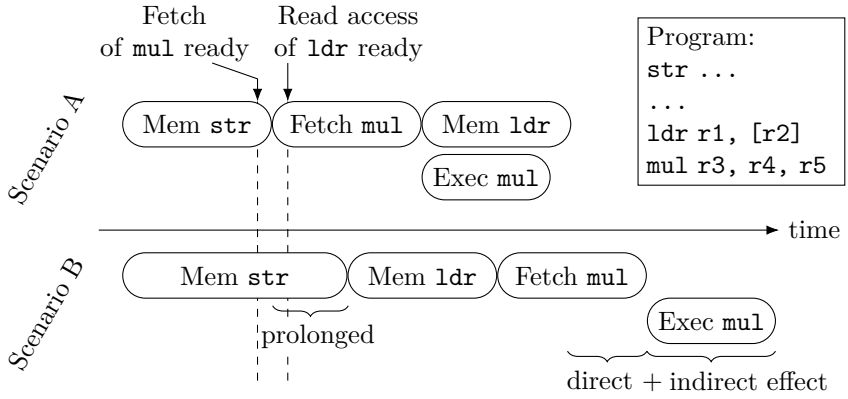


Figure 5.2: Amplifying timing anomaly upon uncertainty of the length of store access `str`, e.g. due to shared-bus blocking.

discussed in Section 4.4 (Non-Monotonicity of In-Order Pipeline), such a pipeline can perform memory accesses out of program order.

In Figure 5.2, we show the program snippet and the amplifying anomaly triggered by the uncertain duration of the store memory access `str`. In Scenario A, the memory access of instruction `str` finishes fast. In Scenario B, the access of `str` is prolonged, e.g. by shared-bus blocking. In both scenarios, the access of instruction `str` is handled by the store buffer in order to not block the actual pipeline. Consequently, the execution of instructions subsequent to `str` can advance in the pipeline.

As a result, the fetch of instruction `mul` becomes ready, but is blocked at first because the memory is busy with `str`. As soon as the write access of `str` finishes in Scenario A, the fetch of `mul` is started as it is the only ready access. During the prolonged write access of `str` in Scenario B, however, the read access of the load instruction `ldr` becomes ready as well. Thus after the access of `str` has finished, there are two ready accesses in Scenario B: the fetch of `mul` and the data access of `ldr`. As data accesses are commonly prioritised over instruction fetches, `ldr` starts first.

In both scenarios, instruction `mul` can only execute after it has been fetched. In Scenario A, `mul`'s execution can be overlapped with the (independent) load. In the five-stage in-order pipeline, the multiplication is

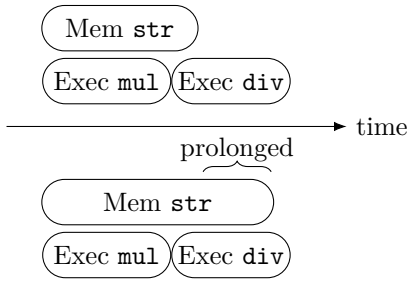


Figure 5.3: The execution of two subsequent arithmetic instructions hiding the latency prolongation of a preceding store.

performed in the execute stage while the load is concurrently performed in the subsequent memory stage. This overlapping is not possible in Scenario B as the load has been performed prior to the `mul` fetch. Ultimately, the incurred penalty on the execution time in Scenario B is larger than the actual prolongation of the write access of `str`. The indirect effect in this example is bounded by the execution latency of `mul` as well as the memory load latency.

Note that an analogous amplifying anomaly can be constructed in case instruction fetches are prioritised over data accesses.

Hiding Latencies Modern processors try to exploit instruction-level parallelism, i.e. they process multiple consecutive instructions at the same time. Consequently, a latency increase of a single instruction can often be (partially) hidden by the execution of other instructions. Thus, the local latency increase is not (fully) visible in the overall response time as seen in Observation 5.1.2.

As an example for our system scenario, consider Figure 5.3. The store operation `str` is performed in the background memory, while subsequent independent arithmetic instructions `mul` and `div` execute concurrently—due to pipelining and the presence of the single-entry store buffer. For the overall execution time, it is irrelevant whether the store is prolonged or not as the memory latency is hidden by the execution of useful independent operations in both cases.

Unlike amplifying timing anomalies, latency hiding effects do not challenge

the soundness of the compositional response time calculation. However, a timing bound that is derived compositionally may overestimate the bound derived by a more integrated analysis such as [Jacobs et al., 2015]. The better a microarchitecture can hide latencies, the less precise will compositionally derived bounds be.

Distinct Worst-Case Paths To a certain extent, latency hiding effects also explain Observation 5.1.3. Another reason for this observation has been briefly mentioned in Section 2.6 (Compositionality): the separate maximisation of the different weight contributions.

In our example scenario, we consider two different weights characterising the execution of a task i: the non-interfered execution time C_i and the memory demand MD_i used to determine the shared-bus interference. Those weights are individually maximised over all execution traces of the program under analysis. The traces exhibiting the respective worst-case behaviour do not have to coincide. Consequently, the trace resulting in the maximal execution time might be less affected by shared-bus interference compared to the trace with maximal memory demand.

By design, the commonly used scheduling interfaces do not take such dependencies into account. Thus, the compositional response time calculation overestimates the response times by combining the individually maximised weight contributions.

Remarks on the Curve Shape In Figure 5.1, the interference response curves feature concave shapes. The reason behind the concave shapes is two-fold. First, we consider the interference at the granularity of individual interfering memory accesses. These individual accesses and their respective blocking behaviour are usually *independent* of each other. Second, in order to maximise the overall response time, the path analysis in [Jacobs et al., 2015] distributes the given interference budget in *decreasing* order w.r.t. the timing impact of the individual interfering accesses. As a consequence, the additional increase in execution time either stays constant or reduces with each additional interfering access.

However, the interference response curves are not guaranteed to be concave. Rare scenarios are conceivable in which an amplifying timing anomaly is only triggered if multiple interfering accesses occur together. Such scenarios would render the curve non-concave.

To summarise, the worst-case response time does not always grow linearly w.r.t. the amount of interference I_j. We illustrated that latency hiding techniques in the microarchitecture as well as the separate maximisation of individual weight contributions can lead to overestimation in a compositional response time calculation. To make matters worse, the compositionality assumption is *invalid*, even for common low-complexity processors, due to the presence of amplifying timing anomalies.

In the following, we discuss the decomposition that underlies the response time formula in Equation 5.1. Thus, we close the gap between the formulas used in schedulability analysis and the formal definition of compositionality (Section 2.6). Afterwards, we propose three approaches to satisfy the compositionality assumption, i.e. how to soundly deal with amplifying timing anomalies. Finally, we provide a qualitative and a quantitative comparison of the approaches in Section 5.6.

5.2 Underlying Decomposition

In the following, we call events that cause an increase in execution time *timing accidents* according to [Wilhelm et al., 2010]. A timing accident is associated with its *penalty* p_{acc}, i.e. the local increase in execution time—its direct effect. Without loss of generality, we consider a single generic source of interference modelled as the number of timing accidents.

The overall goal of timing analysis is to compute a bound on the response times of a given task t, i.e. the number of cycle transitions $w_{time}(\tau)$ in each trace τ through the corresponding execution graph G_t. In order to calculate such bounds, the schedulability analysis composes the ideal execution time C_t and the number of interfering timing accidents I_{acc} obtained from the preceding low-level analyses. The ideal execution time C_t approximates the partially defined weight contribution $wc_{ideal}(\tau)$ that yields the number of cycle transitions in τ if τ does not exhibit a timing accident. I_{acc} bounds the number of timing accidents $wc_{acc}(\tau)$ on a given trace τ. The combination operator $\bigoplus$ is given by

$$\bigoplus(C_t, I_{acc}) := C_t + I_{acc} \cdot p_{acc}.$$

The schedulability analysis is *only* sound if the weights wc_{ideal}, wc_{acc}, and the operator $\bigoplus$ form a max-decomposition in the sense of Definition 2.6.2 in

Section 2.6 (Compositionality). Note that they do not form a decomposition at the level of individual traces as wc_{ideal} is only partially defined.

We define another weight contribution function wc_{base} that counts the number of cycle transitions in a trace *except* for the transitions in which a timing accident is rectified. Each accident is rectified within p_{acc} many cycle transitions, where p_{acc} denotes the direct effect penalty. Note that wc_{base} accounts for any indirect effects caused by an accident. By definition, wc_{base}, wc_{acc}, and $\bigoplus$ form a decomposition at the level of individual traces in the sense of Definition 2.6.1:

$$\forall \text{program } p \, \forall \tau \in \mathcal{T}(G_p).\ w_{time}(\tau) \leq \bigoplus(wc_{base}(\tau), wc_{acc}(\tau)).$$

Unlike wc_{ideal}, an approximation of wc_{base} has to consider the influence of timing accidents in order to account for potential indirect effects. From the perspective of low-level analysis efficiency, there is no gain in approximating wc_{base} rather than w_{time}.

On execution traces that do not contain a timing accident, the weight contributions wc_{base} and wc_{ideal} coincide by definition. If there are *no indirect effects* caused by the considered timing accidents, wc_{ideal} dominates wc_{base} in the sense of Definition 2.5.3 in Section 2.5 (Domination). Consequently, wc_{ideal}, wc_{acc}, and $\bigoplus$ form a max-decomposition.

Theorem 5.2.1. *If wc_{ideal} dominates wc_{base}, the weight contributions wc_{ideal} and wc_{acc} together with the combination operator $\bigoplus$ form a* max-decomposition *of w_{time}.*

Proof. Let a task t be given. We derive

$$\begin{aligned}
&\max_{\tau \in \mathcal{T}(G_t)} w_{time}(\tau) \\
&\leq \max_{\tau \in \mathcal{T}(G_t)} \bigoplus (wc_{base}(\tau), wc_{acc}(\tau)) && \text{Definition} \\
&\leq \bigoplus \left(\max_{\tau \in \mathcal{T}(G_t)} wc_{base}(\tau), \max_{\tau \in \mathcal{T}(G_t)} wc_{acc}(\tau) \right) && \text{Monotonicity of } \bigoplus \\
&\leq \bigoplus \left(\max_{\tau \in \mathcal{T}(G_t)} wc_{ideal}(\tau), \max_{\tau \in \mathcal{T}(G_t)} wc_{acc}(\tau) \right) && \text{Domination} \\
&\leq \bigoplus (C_t, I_{acc}) = C_t + I_{acc} \cdot p_{acc} =: R_t && \text{Analysis}
\end{aligned}$$

□

Finally, we are left with the proof obligation that wc_{ideal} dominates wc_{base}. To prove domination for a concrete type of timing accident and an actual hardware platform, we will show that the sufficient condition of Theorem 2.5.5 in Section 2.5 (Domination) is satisfied. This will either require the absence of indirect effects (Section 5.3) or an adjustment of the penalty used in the combination operator $\bigoplus$ (Section 5.4).

In the following, we consider as timing accidents:

- the prolongation of the memory access latency which might be caused by shared-bus blocking, cache write backs, or DRAM refreshes,
- instruction and data cache misses—both together and in isolation, and
- the handling of processor interrupts.

5.3 Compositionality by Hardware Design

In this section, we revisit our strictly in-order pipeline design from Section 4.5 with a focus on compositionality. Furthermore, we propose a hardware modification which is applicable to any processor and enables compositionality with efficient low-level analysis. For both hardware techniques, we provide formal proofs of domination as demanded in Section 5.2. Note that such custom hardware modifications might currently not be economically feasible. However, they may serve as guidelines for how to design future predictable microarchitectures that allow for rigorous static timing verification.

5.3.1 Stalling

During the rectification of a timing accident in a pipeline stage, an ordinary pipeline greedily continues the execution in the other stages to hide as much of the associated penalty as possible. Due to the greediness, the pipeline might make decisions that actually do not suit the further global execution. As an example, the pipeline might start a long-running and non-preemptive operation such as a memory access which cannot be aborted once the access turns out unnecessary. We call the additional increase in execution time caused by such greedy decisions the *indirect effect* of the timing accident.

We say a pipeline stage *stalls* if it stops the execution of its current instruction, i.e. the stage does not change its state. A common reason for stalls are pipeline hazards, e.g. a data hazard where the decode stage waits for a data memory operation to complete.

Mechanism A mechanism to prevent indirect effects upon a timing accident is to cause *all* pipeline stages to stall. This way, the whole processor core does not change its state during the rectification of a timing accident, e.g. while waiting for the shared bus to become available. Consequently, the core cannot make a decision which additionally prolongs the execution time. On the downside, there is no potential to hide the penalty by overlapped execution of surrounding instructions. Thus, stalling counteracts Observation 5.1.1 and Observation 5.1.2 in Section 5.1.

In order to implement this mechanism in hardware, we introduce a new signal that stalls all pipeline stages except for the stage that incurs the timing accident. The stage that incurs the accident is identified by the signal *stallcore*. The corresponding instruction is allowed to progress within its stage to rectify the timing accident. The signal *stallcore* is generated by the respective hardware component that detects the accident. The cycle behaviour $cycle((stage, cnt))(evs)(p')$ of the in-order pipeline in Appendix A.1 equipped with the stalling mechanism is described as:

$$p' := \lambda i \in \mathcal{I}_d. \begin{cases} (stage'(i), latency(i)) & : stallcore = \bot \\ & \quad \wedge ready(i) \wedge willbefree(stage'(i)) \\ (stage(i), cnt(i)) & : \bot \neq stallcore \neq stage(i) \\ (stage(i), cnt'(i)) & : \text{otherwise} \end{cases}$$

where $\bot$ denotes that the signal *stallcore* is absent, i.e. no timing accident is happening.

Next, we discuss for several types of timing accidents whether the explained mechanism is applicable. If applicable, we prove that wc_{ideal} that ignores traces exhibiting such accidents dominates wc_{base}. We choose $\widehat{C}$, $\widehat{cycle}$, and $\sqsubseteq$ according to Section 3.3. The abstraction $\widehat{C}$ keeps the state of the pipeline's control path concrete while abstracting from the state of the data path. In particular, we thus use the equality of pipeline states as partial order.

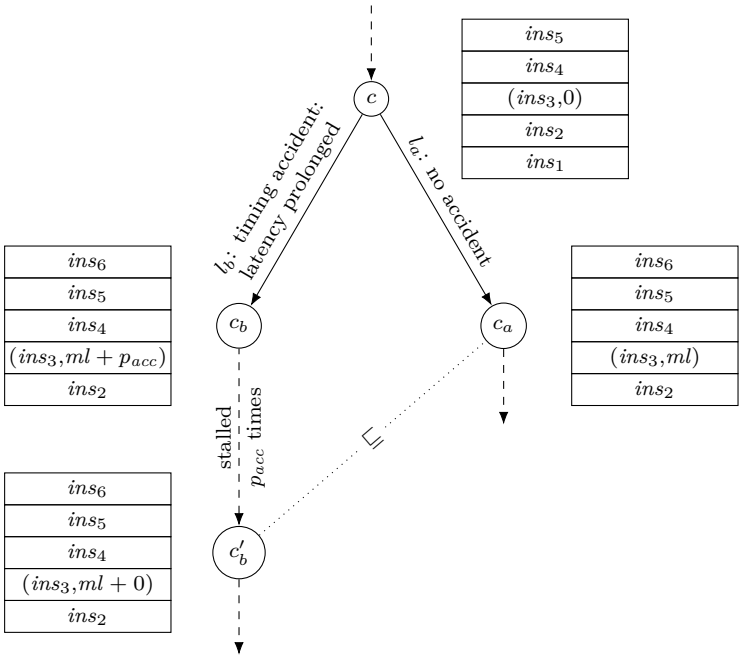


Figure 5.4: wc_{ideal} dominates wc_{base}—or in other words: the no-timing-accident case dominates the prolonged-latency case w.r.t. wc_{base}—when stalling is employed.

Latency Prolongation A group of timing accidents such as shared-bus blocking, cache write backs, or DRAM refreshes has a similar effect: they prolong the latency of an instruction that accesses main memory. The hardware components that encounter the accident, i.e. the shared bus arbiter, the cache controller, or the DRAM controller, signal the processor via *stallcore* that an accident is happening. Consequently, the processor core stalls as long as a timing accident is signalled. This prevents any indirect effects and results in wc_{ideal} dominating wc_{base}.

Lemma 5.3.1. *The weight wc_{ideal} that ignores any execution trace with latency-prolonging timing accidents dominates wc_{base}.*

Proof. We prove the lemma by showing that the sufficient condition of

Theorem 2.5.5 is satisfied. Consider Figure 5.4 as illustration. l_b denotes the presence of a timing accident that prolongs the memory latency by p_{acc}, e.g. a concurrent access blocking the shared bus. l_a denotes the absence of the timing accident. Let a program p, configurations $c, c_b \in \widehat{\mathcal{C}}$ be given such that $\widehat{cycle}(c)(evs_b)(c_b)$ and $l_b \cap evs_b \neq \emptyset$. We choose c_a as the configuration that succeeds c taking the same non-deterministic choices as $\widehat{cycle}(c)(evs_b)(c_b)$ except for the timing accident, i.e. $evs_a \setminus l_a = evs_b \setminus l_b$. Let $\pi^b \in \mathcal{F}_p(c_b)$ be a partial final trace. We choose k as the penalty p_{acc}. We choose $\pi^a = c_a$ to be a path of length zero.

We need to show that $\pi^b_k.c = c'_b \sqsubseteq c_a$ and

$$wc_{base}(c \circ (evs_b, c_b) \circ \pi^b_{0..k}) \leq wc_{ideal}(c \circ (evs_a, c_a)).$$

As the timing accident has only an influence on the latency of the memory operation, the immediate successors c_a and c_b are identical up to the latency of ins_3 in the memory stage. According to the stalling mechanism, the core stalls for the next p_{acc} cycles following c_b. Thus, the resulting configuration c'_b is identical to c_b up to the latency of ins_3 that decreased once per cycle. Consequently, c_a and c'_b are indeed identical and thus $c'_b \sqsubseteq c_a$.

According to our choice of wc_{base}, the cycles between c_b and c'_b which are needed to rectify the accident do not contribute to wc_{base}. We conclude the proof by using the linearity of weights:

$$\begin{aligned} wc_{base}(c \circ (evs_b, c_b) \circ \pi^b_{0..k}) &= wc_{base}(c \circ (evs_b, c_b)) + 0 \\ &= 1 \\ &= wc_{ideal}(c \circ (evs_a, c_a)). \end{aligned}$$ □

Data Cache Miss The cache behaviour of a program is usually already taken into account in the non-interfered execution time wc_{ideal}. In this paragraph, we consider the behaviour of one of the caches in a compositional way. Such a decomposition could for example be desirable for a cumulative analysis of the data cache behaviour. We discuss the behaviour of the data cache in the following, although similar arguments apply for instruction caches as well.

The fetch and memory stage of an in-order pipeline following the von-Neumann architecture scheme share a bus to the common memory that

(ins_5, ml)
ins_4
(`ldr`, 0)
ins_2
ins_1

Figure 5.5: Pipeline state that leads to indirect effects caused by `ldr` missing the data cache.

holds code and data. An additional data cache miss can trigger significant indirect effects.

As an example, consider the execution time of instruction ins_4 in the pipeline state depicted in Figure 5.5. If the load `ldr` hits the data cache, instruction ins_4 leaves the pipeline after four cycles assuming the absence of further hazards. If the load `ldr` misses the data cache, the load instruction and consequently ins_4 are stalled because the memory bus is blocked by the instruction fetch of ins_5. The data memory access of `ldr` starts as soon as the fetch of ins_5 finishes. As soon as the load access completes, instruction ins_4, which occupied the execute stage in the meantime, needs three cycles to leave the pipeline. The overall timing difference amounts to $2 \cdot ml - 1$, which is almost twice the direct effect penalty for a cache miss—namely the memory latency ml. Even worse, the pipeline states resulting from the hit and miss case are incomparable according to the chosen partial order $\sqsubseteq_{\equiv=}$.

The stall mechanism described above cannot be applied in this particular case. As a data cache miss might need to wait for a fetch to complete, the fetch stage cannot stall during this timing accident without deadlocking the core. Stalling the core only when the cache miss has access to the bus is not sufficient. The resulting states are still incomparable as the pipeline advances while waiting for the bus to become free.

Instruction and Data Cache Miss In a preemptive setting, evictions of a preempting task can cause additional cache misses in the preempted task. As the cache-related preemption delay depends on scheduling decisions, it is considered during the high-level schedulability analysis. This requires a decomposition into non-preempted execution and additional cache misses

due to preemption. In this paragraph, we consider *both* data and instruction caches in a compositional manner at the same time.

The stalling mechanism causes the processor core to stall while an instruction or data cache miss is rectified, i.e. a cache line is transferred from main memory to the respective cache. Consequently, a memory access resulting from a cache miss can never be blocked by an ongoing access to the common bus. The only corner case arises if two instructions miss both caches, instruction and data, at the same time. The requested cache lines are loaded one after another taking $2 \cdot ml$ cycles, i.e. the direct effect penalty for a data miss plus the penalty for an instruction miss. During this time, the processor core is stalled resulting in the same pipeline state as if both instructions hit the caches.

Thus, the weight wc_{ideal}, which is only defined on execution traces exhibiting cache hits exclusively, dominates the weight wc_{base}. Therefore, the efficient low-level analysis can be employed that approximates wc_{ideal} and only follows the cache hit cases. The formal proof is along the lines of the proof for the latency prolongation accident (see Figure 5.4).

Interrupts The stalling mechanism to ensure compositionality is not compatible with interrupts. Upon an interrupt, the pipeline stops the execution of the current program in order to rectify the timing accident, i.e. to execute the interrupt service routine. As soon as the service routine completes, the execution of the original program is resumed. Since the rectification requires the pipeline, stalling the pipeline upon an interrupt cannot be a solution.

5.3.2 Strictly In-Order Pipeline

In Section 4.5, we introduced a strictly in-order pipeline which we designed with predictability in mind. Indeed, we have been able to demonstrate in Section 4.7 that our progress-based abstraction does not exhibit anomalous behaviour w.r.t. w_{time}, i.e. a low-level analysis could safely follow the locally worst case only. In this section, we investigate the influence of the strictly in-order design principle on compositionality and whether we can prove the absence of amplifying timing anomalies.

The progress-based partial order $\sqsubseteq$ offers more flexibility in the domination proofs than the equality of pipeline states. This increased flexibility allows to partially overlap timing accidents with the execution of other

instructions (cf. Observation 5.1.2)—in contrast to the above stalling mechanism. We can still prove that wc_{ideal} dominates wc_{base}. This is important to enable an efficient low-level analysis that can ignore the timing accident cases.

Latency Prolongation States of the strictly in-order pipeline are partially ordered according to the progress of instructions within the pipeline. The more progress a pipeline state has, the shorter the remaining execution time to finish the current program. Furthermore, the progress of a pipeline state can only increase by a cycle transition. Consequently, after the rectification of a timing accident, the respective pipeline state has at least the same progress as the state without the accident.

The strictly in-order design prevents any indirect effects and results in wc_{ideal} dominating wc_{base}.

Lemma 5.3.2. *The weight wc_{ideal} that ignores any execution trace with latency-prolonging timing accidents dominates wc_{base}.*

Proof. We prove the lemma by showing that the sufficient condition of Theorem 2.5.5 is satisfied. l_b denotes the presence of a timing accident that prolongs the memory latency by p_{acc}, e.g. a concurrent access blocking the shared bus. l_a denotes the absence of the timing accident. Let a program p, configurations $c, c_b \in \widehat{\mathcal{C}}$ be given such that $\widehat{cycle}(c)(evs_b)(c_b)$ and $l_b \cap evs_b \neq \emptyset$. We choose c_a as the configuration that succeeds c taking the same non-deterministic choices as $\widehat{cycle}(c)(evs_b)(c_b)$ except for the timing accident, i.e. $evs_a \setminus l_a = evs_b \setminus l_b$. Let $\pi^b \in \mathcal{F}_p(c_b)$ be a partial final trace. We choose k as the penalty p_{acc}. We choose $\pi^a = c_a$ to be a path of length zero. Figure 5.6 illustrates such a scenario.

We need to show that $\pi^b_k.c = c'_b \sqsubseteq c_a$ and

$$wc_{base}(c \circ (evs_b, c_b) \circ \pi^b_{0..k}) \leq wc_{ideal}(c \circ (evs_a, c_a)).$$

As the timing accident has only an influence on the latency of the memory operation, the immediate successors c_a and c_b are identical up to the latency of ins_3 in the memory stage. According to the strictly in-order design, the pipeline progresses from c_b to $c'_b \sqsubset c_b$ during the p_{acc} many cycles, Lemma 4.6.3 (Positive Progress). Furthermore, the latency of memory operation ins_3 decreases from $ml + p_{acc}$ to ml in the meantime. Consequently, $c'_b \sqsubseteq c_a$.

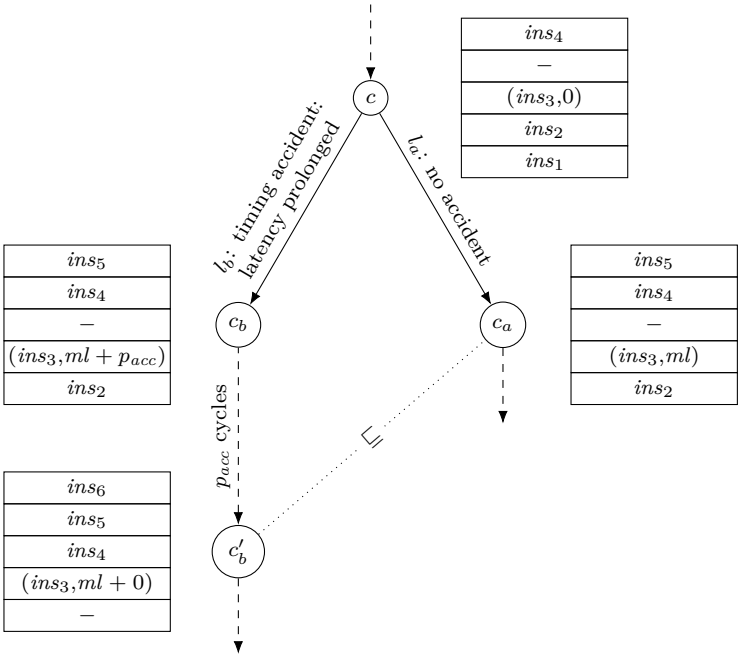


Figure 5.6: wc_{ideal} dominates wc_{base} in our strictly in-order design, or in other words: the case without timing accident dominates the prolonged-latency case w.r.t. wc_{base}. In contrast to the stalling mechanism, all instructions can make progress during the rectification of the timing accident.

According to our choice of wc_{base}, the cycles between c_b and c'_b needed to rectify the accident do not contribute to wc_{base}. We conclude the proof by using the linearity of weights:

$$\begin{aligned} wc_{base}(c \circ (evs_b, c_b) \circ \pi^b_{0..k}) &= wc_{base}(c \circ (evs_b, c_b)) + 0 \\ &= 1 \\ &= wc_{ideal}(c \circ (evs_a, c_a)). \end{aligned}$$ □

Data Cache Miss Unlike ordinary in-order pipelines, the strictly in-order design guarantees that the common bus is free if an instruction misses the data cache. In particular, a data cache miss cannot be blocked by an ongoing fetch of a succeeding instruction.

While the cache miss is served, i.e. the requested cache line is loaded from main memory, the pipeline can only progress further according to Lemma 4.6.3 (Positive Progress). This precludes any indirect effects caused by a data cache miss. The formal domination proof follows the lines of the proof for the latency-prolongation accident.

Instruction Cache Miss For instruction cache misses, the situation is more complicated compared to data cache misses. In order to prevent reordering effects on the common bus, the strictly in-order pipeline stalls instruction cache misses as long as preceding instructions might perform data accesses on the bus. Besides the memory latency *ml* needed to fetch a cache line, an instruction cache miss causes additional pipeline stalls while waiting for data memory instructions to finish. Thus, an instruction cache miss can cause *indirect effects*. Note that these indirect effects occur no matter whether the preceding memory instructions hit or miss the data cache.

Consequently, wc_{ideal} does *not* dominate wc_{base} w.r.t. instruction cache misses. The weight contributions wc_{ideal} and wc_{acc} together with the combination operator $\bigoplus$ that employs the direct effect penalty, do *not* constitute a decomposition. Nevertheless, the two following sections on *sound penalties* and the *compositional base bound* provide solutions to compositionally account for the effects of instruction cache misses in a strictly in-order pipeline design.

Interrupts Interrupts in the strictly in-order pipeline can be accounted for in a compositional manner. In Section 5.4.2, we provide details on how to derive a sound penalty for this timing accident.

5.4 Compositionality by Sound Penalty

Consider the decomposition presented in Section 5.2. Our goal is still to show that wc_{ideal} dominates wc_{base}, where wc_{base} accounts for all additional cycles caused by a timing accident except for those accounted by the penalty p_{acc} used in $\bigoplus$. As a consequence, the results obtained by an efficient low-level analysis approximating wc_{ideal}, i.e. assuming the absence of timing accidents, could be safely used within a compositional schedulability analysis.

Up to now, we assumed that the penalty p_{acc} covers the direct effect of a timing accident, i.e. the time needed to rectify it. If an accident can, however, cause indirect effects, the above statement does not hold any longer. Consequently, no analysis component—neither wc_{ideal} nor wc_{acc} nor $\bigoplus$—will account for the indirect effect. One possibility to remedy this situation is to adjust the penalty p_{acc} associated with the timing accident such that the penalty, and thus $\bigoplus$, accounts for *all* possible indirect effects caused by a single occurrence of the respective accident—additionally to the direct effect. Using such a penalty, wc_{ideal} trivially dominates wc_{base}. While this approach tackles the soundness issue in Observation 5.1.1, it aggravates the imprecision found in Observation 5.1.2.

Given a microarchitecture, finding such a penalty that is valid under all circumstances is difficult. It requires an in-depth analysis of the microarchitectural behaviour to determine the maximal indirect effect triggered by the respective timing accident. To the best of our knowledge, there is no (automated) technique known to bound the maximal indirect effect. A part of this problem is to identify whether a microarchitecture exhibits *domino effects*. In this case, the indirect effect is not bounded at all and thus no sound penalty can be found.

For the strictly in-order pipeline with its progress-based partial order, the problem is less complicated. In case of a timing accident, after some fixed amount of cycles, the respective configuration has more progress than the immediate successor in the non-accident case. Due to monotonicity, we know that the indirect effects are bounded by this amount of cycles. We provide examples below that illustrate how sound penalties for instruction

cache misses and interrupts can be derived for the strictly in-order pipeline design.

For general microarchitectures, we are not able to derive penalties that capture indirect effects experienced by *any* program. For an individual program, however, we can compute its interference response curve. Based on its slope, we can derive an upper bound on the indirect effects of a timing accident experienced by that particular program. A sound, program-independent penalty would have to be at least as high.

5.4.1 Per-program Sound Penalties

We can experimentally derive sound penalties that account for the indirect effects of a timing accident for a particular program. We use these program-specific penalties in the experimental evaluation in Section 5.6 to evaluate the sound penalty approach. However, the overestimation turns out to be too excessive to obtain meaningful results. Furthermore, the computational effort to calculate the program-specific penalties is significant. Thus, we present this approach only for the sake of completeness.

In order to derive a penalty specific to a given program, we use the integer linear program that encodes the interference response curve of Section 5.1. This linear program is generated as part of the low-level analysis proposed in [Jacobs et al., 2015]. We present the formulation of the ILP for shared-bus interference, but it applies in a similar way to other sources of interference as well.

First, we compute an upper timing bound t_0 assuming no interference. This bound t_0 corresponds to the y-intercept of the interference response curve. Next, we introduce an integer variable i to model a variable amount of interference and change the corresponding interference constraint (cf. Section 3.3):

$$\sum_{e \in \widehat{E}_p} \mathit{blocked}^{lb}(e) \cdot x_e \leq i.$$

The curve itself is described by the maximisation objective:

$$\sum_{e \in \widehat{E}_p} \mathit{time}^{ub}(e) \cdot x_e.$$

We are searching for the smallest penalty p, such that the linear function $t_0 + p \cdot i$, which corresponds to the compositional combination operator,

over-approximates the interference response curve. As i is already a variable, p cannot be a variable as well because the resulting term $p \cdot i$ is no longer linear and we could not use our linear solver for this problem.

However, we can conjecture a constant penalty p_c and use an ILP formulation to check whether p_c is sufficient to capture all indirect effects. To this end, we add the following constraint:

$$\sum_{e \in \widehat{E}_p} time^{ub}(e) \cdot x_e > t_0 + p_c \cdot i.$$

As we only need to check for feasibility of the above ILP, the objective function is irrelevant. If the above ILP is feasible, the solver has found an amount of interference—given by the valuation of i—such that the linear function is below the interference response curve. Thus, the conjectured penalty p_c was not sufficient. If the above ILP is infeasible, we know that p_c was sufficient to guarantee that the linear function $t_0 + p_c \cdot i$ is above the interference response curve for all i.

We employ this check in a binary search to obtain a sound penalty for the given program. We start with a lower bound l and an upper bound u such that the ILP is feasible for $p_c = l$ and infeasible for $p_c = u$. Next, we perform the above ILP check with $p_c = \lceil \frac{l+u}{2} \rceil$. If the ILP is feasible, we found a counterexample and can refine l. If the ILP is infeasible, we have a sound penalty and can refine u. We repeat the search until the difference between l and u is below a certain precision threshold, e.g. 0.001.

Note that the approach to calculate a sound per-program penalty requires the less efficient low-level analysis from [Jacobs et al., 2015] that takes the potential interference caused by the respective timing accident already into account.

5.4.2 Strictly In-Order Pipeline

At the end of Section 5.3.2, we have seen that the direct effect penalty is not enough to handle instruction cache misses in a compositional way. In the next paragraph, we calculate a sound penalty that incorporates any indirect effects caused by an instruction cache miss in the strictly in-order pipeline.

Additionally, we describe how interrupts can be soundly decomposed from uninterrupted execution time. This involves the computation of upper

bounds on the interrupt latency—the time between an active interrupt signal and the start of the service routine—and the restart latency—the time needed after the termination of the service routine to reach a pipeline state with at least as much progress as before the interrupt.

Instruction Cache Miss An instruction cache miss can cause indirect effects in a strictly in-order pipeline design. In this paragraph, we compute an upper bound on the indirect effects in terms of clock cycles. Adding this upper bound to the direct effect penalty results in a sound penalty for an instruction cache miss. Using this sound penalty, the weight wc_{ideal} trivially dominates wc_{base} which allows for an efficient low-level analysis.

We provide two ways of bounding the indirect effects related to an instruction cache miss. Consider Figure 5.7. In configuration c, instruction ins_6 is about to be fetched and either hits or misses the instruction cache, resulting in c_a and c_b. In the cache miss case, the fetch is blocked until all data memory operations reach the end of the memory stage (configuration c'_b). In the worst case, all preceding instructions ins_3, ins_4 and ins_5 are data-dependent and perform memory operations that access the common bus. Thus, it might take up to $3 \cdot ml + 3$ cycles until ins_5 has finished its access and the fetch of ins_6 can start. The fetch of ins_6 itself takes $ml + 1$ cycles—one cycle to start the access and ml many cycles to transfer the cache line—resulting in configuration c''_b. We know that $c''_b \sqsubseteq c_a$: ins_6 has the same progress in both configurations, and the preceding instructions have progressed in c''_b at least as much as in c_a because they do not depend on ins_6 (see Lemma 4.6.2 (Progress Dependence)). Consequently, $4 \cdot ml + 4$ is a sound penalty, taking into account direct and indirect effects.

However, the penalty $4 \cdot ml + 4$ is quite pessimistic. It comprises the cycles required to execute ins_3, ins_4 and ins_5 between c_b and c'_b. These cycles will also occur in the cache hit case after c_a. If these instructions take many cycles to complete, they will eventually block the execution of later instructions $ins_6, ins_7, \ldots$ in the hit case due to the limited pipeline length. To put it in a nutshell, only the *overlapping* of the execution of $ins_6, ins_7, \ldots$ with the execution of ins_3, ins_4, ins_5 in the hit case is prevented in the miss case. Thus, an upper bound on the indirect effects should only account for this overlapping.

This observation leads us to the second way of bounding the indirect effects. We consider the further execution from c_a and c''_b. Let c'_a de-

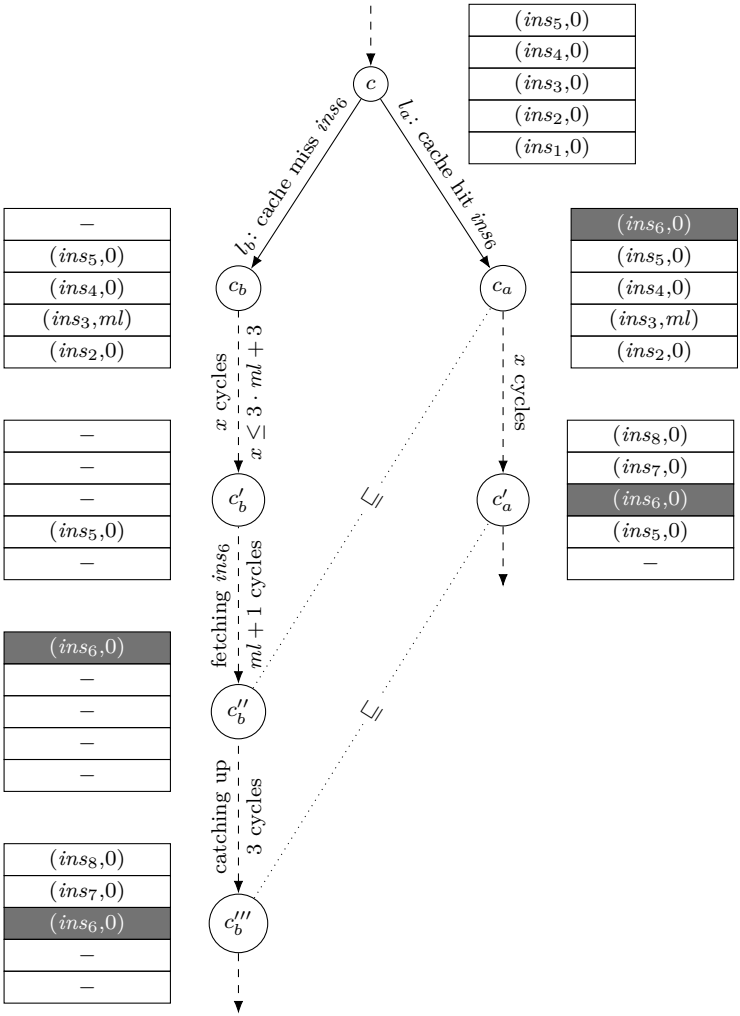


Figure 5.7: Deriving a sound penalty for an instruction cache miss of instruction ins_6 in our strictly in-order design. ins_6 is the instruction to fetch next and the fetch stage becomes available for ins_6 in the cycle following c, i.e. all instructions $ins_5, ins_4, ins_3, ins_2$, and ins_1 are ready in c.

note the configuration reached from c_a by executing ins_3, ins_4 and ins_5 as long as they could block an instruction memory access. The progress of instructions up to ins_5 in c'_a matches the progress of those instructions in c'_b. Furthermore, c'_a exhibits the maximal progress that $ins_6, ins_7, \ldots$ can reach while executing in an overlapped fashion with the earlier instructions. Now, consider the successor configurations of c''_b until they catch up with the progress of c'_a. The number of cycles between c''_b and c'''_b corresponds to the time ins_6 needs to progress from $(IF, 0)$ to $(EX, 0)$. Assuming a maximal latency of two cycles for an instruction in the execute stage, the pipeline needs at most three cycles to catch up with the hit case. Overall, the effective indirect effect cannot exceed four cycles. Summing up the individual parts leads, due to the linearity of weights, to a sound penalty of $ml + 4$ cycles that accounts for direct and indirect effects.

The penalty $ml + 4$ assumes that the overlap of the execution of instructions $ins_6, ins_7, \ldots$ with the execution of ins_3, ins_4, and ins_5 amounts to a maximum of three cycles. Variants of our strictly in-order pipeline could feature techniques to complete instructions in early stages, e.g. the early execution of unconditional branches or nop instructions. If the subsequent instructions $ins_6, ins_7, \ldots$ are unconditional branches, they could execute and complete in the fetch or decode stage in parallel to and independently of ins_3, ins_4, ins_5. The maximal amount of overlap in this case would only be bounded by the latencies of ins_3 to ins_5. Consequently, for such architectural variants, we need to resort to the conservative penalty $4 \cdot ml + 4$ which we derived first.

Interrupts Interrupts are used to handle events asynchronously to the execution of user programs. As an example, periodic timer interrupts are used to regularly invoke the operating system in order to implement preemptive scheduling. As described in Section 3.4 (Scheduling Interface and Compositionality), there is a pipeline-related timing penalty associated with the handling of interrupts: the interrupt latency—the time between an active interrupt signal and the start of the service routine—and the restart latency—the time needed after the termination of the service routine to reach a pipeline state with at least as much progress as the state before the interrupt. In this paragraph, we calculate an upper bound on the pipeline-related interrupt cost for our strictly in-order pipeline design.

Before we derive the interrupt cost, we describe an implementation of the

interrupt mechanism. While servicing interrupts, the core is in a privileged mode that has unlimited access to all resources such as external devices. To cleanly separate execution in privileged and normal user mode, we do not allow to overlap instructions that should be executed in different modes. There are two contrary optimisation goals w.r.t. the timing of (external) interrupts, e.g. caused by a timer component. On the one hand, the interrupt should be serviced as soon as possible which requires to flush (parts of) the pipeline, i.e. the currently executed instructions of the user mode program. On the other hand, flushed instructions need to be restarted after the interrupt resulting in work, such as e.g. fetch or decode, being done twice. Our implementation completes instructions in the memory and write-back stages and flushes all other instructions upon an external interrupt. Furthermore, there are operations such as memory accesses that are not interruptible and thus need to completed before jumping to the interrupt service routine.

We need to calculate a sound penalty such that wc_{ideal}, which is only defined on execution traces without interrupts, dominates wc_{base}. In this calculation, we want to exploit the progress-based partial order $\sqsubseteq$ introduced in Chapter 4 (Progress-based Abstraction). We discussed in Section 4.9 (Outlook: Monotonic Extensions) that the strictly in-order pipeline equipped with an interrupt mechanism as sketched above does *not* behave monotonically and thus precludes the use of the progress-based order $\sqsubseteq$. However, consider an (abstract) execution graph where interrupts are modelled explicitly as non-deterministic choices (Figure 5.8). The goal is to decompose the interrupt cases from the execution graph: i.e. to remove the interrupt-related execution parts from the graph and to add the interrupt-related cost to the composition operator. Assume we decompose the interrupt cases step-by-step in a backwards manner from the end to the start of the given execution graph. As a consequence, we can use $\sqsubseteq$ defined for the strictly in-order pipeline *without* interrupts in each single step as no interrupt occurs in the further execution, respectively. Thus, despite the non-monotonicity introduced by interrupts, we can still use $\sqsubseteq$ to calculate sound bounds on the interrupt and restart latencies.

We are now left with calculating upper bounds on the interrupt and restart latency. Using them as timing penalty, the non-interrupt case dominates the interrupt case w.r.t. wc_{base}, or in other words wc_{ideal} dominates wc_{base}. This enables an efficient low-level analysis that considers only uninterrupted task executions.

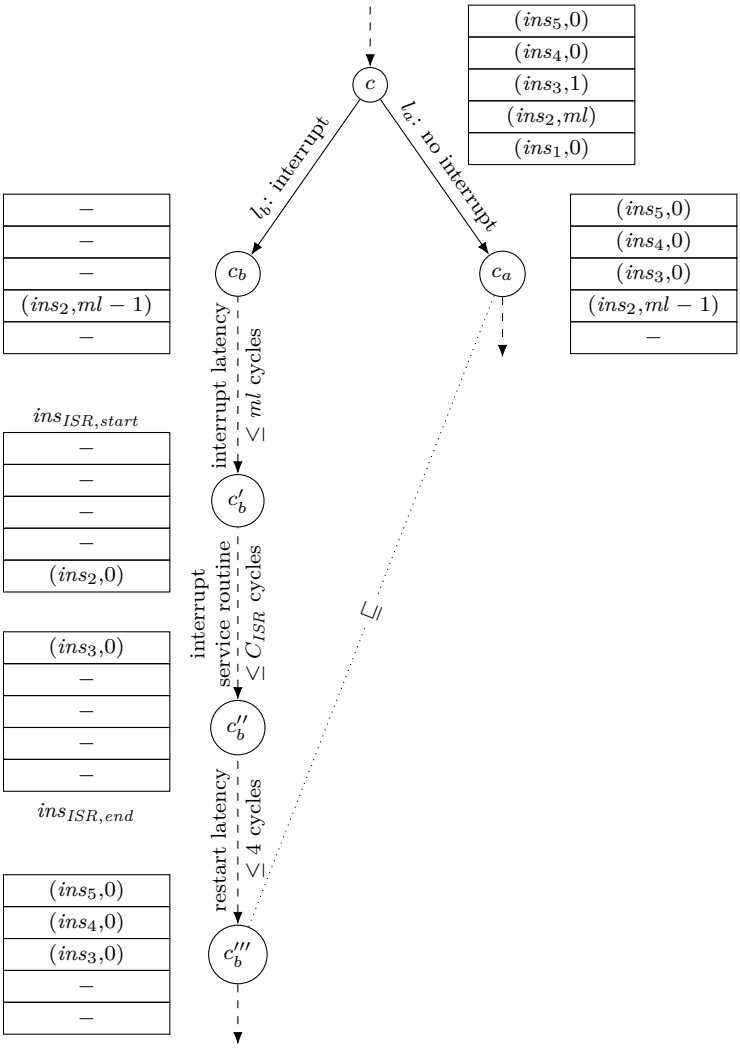


Figure 5.8: Deriving a sound penalty for a single interrupt in our strictly in-order design. The configuration c depicts only one possible example. To derive a sound penalty, we need to consider all possible configurations that could encounter an interrupt.

Consider Figure 5.8 which depicts the timing behaviour of an interrupt triggered in an example configuration c. The worst interrupt latency is caused by a memory instruction that has just started before the interrupt signal becomes active. The pipeline has to wait up to $ml - 1$ cycles for the memory access to finish, and—in case of a data memory access—another cycle to advance the instruction to the write-back stage.

The resulting configuration is an *initial* configuration for the interrupt service routine. The following cycles, until the last instruction of the ISR has left the pipeline, are accounted for by the execution time weight of the interrupt service routine. These cycles do not contribute to the interrupt-related pipeline cost and are upper bounded by C_{ISR} that is determined by a separate low-level analysis.

The restart latency covers the cycles needed to reach a state with at least as much progress as in c_a, the non-interrupt case. As instructions ins_1 and ins_2 finish before the interrupt service routine is executed, they have the maximally possible progress afterwards. Instructions $ins_3, ins_4, \ldots$ have not progressed further than $(EX, 0)$ in c before the interrupt signal became active. In configuration c_a, the maximal progress of these instructions is thus (MEM, l) where l is the latency associated to ins_3. How many cycles does it take to reach at least the progress of *any* possible configuration c_a? A configuration c_a without pipeline bubbles—i.e. with the maximal number of ongoing instructions—is the worst configuration w.r.t. the restart latency. We assume that all instruction fetches are cache hits since they have been loaded into the cache before the interrupt. The cache-related preemption delay accounts for the potential cache reloads in case the execution of the service routine evicts useful cache blocks. The maximal restart latency amounts to 4 cycles in order to advance instruction ins_3 from $(IF, 0)$ to (MEM, l)—assuming a maximal execution latency of two cycles in stage EX.

To conclude, the combination operator $\bigoplus$ needs to account for additional $ml + 4$ cycles per interrupt as interrupt-induced pipeline overhead. Then, wc_{ideal} dominates wc_{base} and a sound low-level analysis can assume the absence of interrupts.

5.5 Compositional Base Bound

In Section 5.4, we presented one possibility to deal with indirect effects, namely to incorporate them into the penalty part of the decomposition.

In that case, wc_{ideal} dominates wc_{base} and a sound low-level analysis can assume the absence of the respective timing accidents. The approach has the advantage that the overall analysis is computationally efficient. However, the approach attributes the maximally possible indirect effect to *each occurrence* of the respective timing accident which introduces severe pessimism. In practice, indirect effects occur only rarely and under certain circumstances.

Another possibility to deal with indirect effects is to account for them during a low-level analysis that approximates wc_{base}. On the one hand, we do not longer require that wc_{ideal} dominates wc_{base}. On the other hand, such an analysis cannot assume the absence of timing accidents but has to analyse their (indirect) effect on the execution time. Thus, we trade off the efficiency of the individual low-level analyses against precision: we account for indirect effects only when they can actually happen and not at each occurrence of an accident. The program characteristics which is obtained by the low-level analysis approximating wc_{base} accounts for possible indirect effects. We term this characteristic the *compositional base bound* B_p of a program p. The compositional base bound captures all processor cycles that are not explained by the direct effect of timing accidents. The compositional base bound B_p can be used as a replacement for the non-interfered execution time bound C_p in existing schedulability analyses such as presented in Section 3.5. This enables the sound timing analysis of systems that exhibit indirect effects, i.e. amplifying anomalies and even domino effects (cf. Observation 5.1.1).

The computation of the compositional base bound for a given program is based on the integer linear program that encodes its interference response curve. This linear program is generated as part of the low-level analysis that takes the effect of timing accidents into account. Jacobs et al. [2015] proposed such an analysis for shared-bus interference.

Recall the integer linear program used for path analysis described in Section 3.3. We introduce an additional integer-valued variable i to model a variable amount of interference caused by the considered timing accidents. We change the corresponding interference constraint, e.g. for shared-bus interference, by replacing the constant amount of shared-bus interference I by the new variable i:

$$\sum_{e \in \widehat{E}_p} blocked^{lb}(e) \cdot x_e \leq i.$$

The interference response curve is implicitly given by the objective

$$\max \sum_{e \in \widehat{E}_p} time^{ub}(e) \cdot x_e.$$

We obtain the compositional base bound by maximising

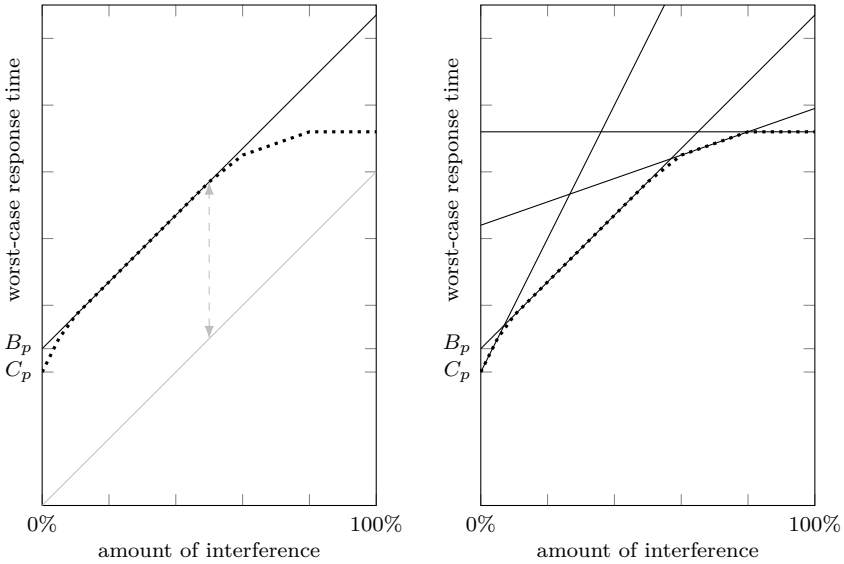
$$\max \left(\sum_{e \in \widehat{E}_p} time^{ub}(e) \cdot x_e \right) - pn \cdot i,$$

where pn denotes the direct effect penalty.

The first term captures the entire execution time of the program, including interference by timing accidents. The second term, $pn \cdot i$, captures the share of execution time that is explained by the direct effects of i occurrences of the respective timing accident. Thus, the difference between the two terms captures the share of execution time not explained by the direct effects of interference. This includes the ideal execution time but also indirect effects caused by timing accidents. By maximising over all possible amounts of interference i, the solution to the above ILP provides an upper bound on the compositional base bound. The compositional base bound includes the maximum possible indirect effects for the program under analysis.

The compositional base bound can also be interpreted graphically. Consider the interference response curve in Subfigure 5.9a. Imagine a linear curve with slope equal to the direct effect penalty pn. Now, move this linear curve vertically until the curve is fully above the interference response curve of interest. The y-intercept of the linear curve then corresponds to the compositional base bound computed by the above integer linear program.

The compositional base bound approach is not limited to the direct effect penalties. The above ILP formulation provides a sound base bound that satisfies the compositional assumption for *any* penalty. Using a lower penalty than the direct effect penalty can be useful to obtain tighter approximations in the regions where the interference response curve levels off. In the extreme case $pn = 0$, the compositional base bound corresponds to an interference-insensitive bound. Consider Subfigure 5.9b for an illustration. The use of lower penalties tackles the potential precision loss found in Observations 5.1.2 and 5.1.3. To increase overall precision, the interface between low-level and schedulability analysis could be enhanced to allow



(a) Graphical interpretation of the compositional base bound calculation.

(b) Improved precision by multiple compositional base bounds using different penalties.

Figure 5.9: Sound linear approximations of a given interference response curve (dashed) using the compositional base bound approach. C_p denotes the execution time of program p in isolation, B_p denotes the compositional base bound using the direct effect penalty.

for multiple linear approximations with different penalties. The high-level schedulability analysis can finally pick the lowest value among all sound, linear approximations.

The compositional base bound approach is not limited to a single type of timing accidents causing interference. In practice, there are multiple different sources of interference, e.g. cache reloads due to preemption, shared-bus blocking, and DRAM refreshes. The above ILP formulation can be extended to account for multiple sources of interference. Let $i_1, i_2, \ldots, i_n$ be integer-valued variables such that i_k models the variable amount of interference caused by timing accidents of type k. In the corresponding interference constraints of the implicit path enumeration, we use these variables instead of constant values for the respective amount of interference. The objective function to compute the compositional base bound becomes

$$\max \left(\sum_{e \in \widehat{E}_p} time^{ub}(e) \cdot x_e \right) - \sum_{k=1}^{n} pn_k \cdot i_k,$$

where pn_k denotes the (direct effect) penalty associated with timing accidents of type k. Note that the runtime of the low-level analysis which accounts for the indirect effects can increase significantly with the number of different types of timing accidents.

Next, we prove the correctness of the compositional base bound approach relative to the low-level analysis used to compute the interference response curve.

Theorem 5.5.1. *Let n be the number of different timing accidents that affect the execution time. Let $irc(\boldsymbol{I})$ be the value of the interference response curve for a given amount of interference $\boldsymbol{I} = (I_1, \ldots, I_n)$ and b the respective compositional base bound as computed above. The compositional base bound can be used to bound the interference response curve:*

$$\forall \boldsymbol{I}.\, irc(\boldsymbol{I}) \leq b + \sum_{k=1}^{n} pn_k \cdot I_k = \bigoplus(b, \boldsymbol{I}).$$

Proof. Let $irc(\boldsymbol{I})$ be the value of the interference response curve for a given amount of interference $\boldsymbol{I}$, i.e.:

$$\begin{aligned} irc(\boldsymbol{I}) = \max_{\boldsymbol{x}} & \sum_{e \in \widehat{E}_p} time_e \cdot x_e, \\ \text{s.t. } & A \cdot \boldsymbol{x} \leq \boldsymbol{I}, \\ & B \cdot \boldsymbol{x} \leq \boldsymbol{D}, \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

where $\boldsymbol{x}$ is the vector of frequency variables x_e, $\boldsymbol{D}$ and $\boldsymbol{I}$ are vectors of integer-valued coefficients, and A and B are integer-valued matrices, such that $A \cdot \boldsymbol{x} \leq \boldsymbol{I}$ encodes the interference constraints and $B \cdot \boldsymbol{x} \leq \boldsymbol{D}$ encodes flow constraints and loop bounds. More details on the constraints can be found in Section 3.3.

The compositional base bound b is obtained by the following ILP:

$$\begin{aligned} b = \max_{\boldsymbol{i},\boldsymbol{x}} & \sum_{e \in \widehat{E}_p} time_e \cdot x_e - \sum_{k=1}^{n} pn_k \cdot i_k, \\ \text{s.t. } & A \cdot \boldsymbol{x} \leq \boldsymbol{i}, \\ & B \cdot \boldsymbol{x} \leq \boldsymbol{D}, \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

where $\boldsymbol{x}, A, B$, and $\boldsymbol{D}$ *are the same* as in the previous ILP and $\boldsymbol{i}$ is a vector of additional integer-valued variables—one per type of timing accident.

We need to show that for any amount of interference $\boldsymbol{I}$, we have

$$b + \sum_{k=1}^{n} pn_k \cdot I_k \geq irc(\boldsymbol{I}),$$

which is equivalent to $b \geq irc(\boldsymbol{I}) - \sum_{k=1}^{n} pn_k \cdot I_k$.

Due to the maximisation over $\boldsymbol{i}$, we have for any $\boldsymbol{I}$,

$$\begin{aligned} b \geq \max_{\boldsymbol{x}} & \left(\sum_{e \in \widehat{E}_p} time_e \cdot x_e \right) - \sum_{k=1}^{n} pn_k \cdot I_k, \\ & \text{s.t. } A \cdot \boldsymbol{x} \leq \boldsymbol{I}, \\ & \quad B \cdot \boldsymbol{x} \leq \boldsymbol{D}, \\ & \quad \boldsymbol{x} \geq \boldsymbol{0} \\ = irc(\boldsymbol{I}) & - \sum_{k=1}^{n} pn_k \cdot I_k. \end{aligned}$$

□

Corollary 5.5.2. *Let w_{time} denote the number of cycle transitions on a given execution trace, and the vector $\boldsymbol{wc}_{acc}$ denote the number of timing accidents with respective penalties $\boldsymbol{pn}_{acc}$. The base weight contribution $wc_{base}(\tau)$ is given as $w_{time}(\tau) - \boldsymbol{wc}_{acc}(\tau) \cdot \boldsymbol{pn}_{acc}$, counting all cycle transitions not covered by the given penalties. Given the analysis computing irc is sound for every task t, i.e.*

$$\forall \tau \in \mathcal{T}(G_t).\, w_{time}(\tau) \leq irc(\boldsymbol{wc}_{acc}(\tau)),$$

the adjusted analysis computing the compositional base bound b is sound, i.e.

$$\max_{\tau \in \mathcal{T}(G_t)} wc_{base}(\tau) \leq b.$$

Proof. Let τ be the execution trace that maximises wc_{base}.
We derive

$$\begin{aligned} wc_{base}(\tau) &= w_{time}(\tau) - \boldsymbol{wc}_{acc}(\tau) \cdot \boldsymbol{pn}_{acc} && \text{Definition} \\ &\leq irc(\boldsymbol{wc}_{acc}(\tau)) - \boldsymbol{wc}_{acc}(\tau) \cdot \boldsymbol{pn}_{acc} && \text{Assumption} \\ &\leq b && \text{Theorem 5.5.1} \end{aligned}$$

□

Efficient Computation

Last, we make a remark on the efficiency of the compositional base bound calculation. In order to obtain a reasonable compositional base bound, the abstract execution graph distinguishes edges that differ w.r.t. $blocked^{lb}$. This distinction negatively affects the size of the execution graph and thus the size of the linear program to be solved. Consider the ILP formulation to compute the compositional base bound. The objective is maximised if and only if

$$i = \sum_{e \in \widehat{E}_p} blocked^{lb}(e) \cdot x_e.$$

We can substitute i within the objective function and obtain by associativity

$$\max \sum_{e \in \widehat{E}_p} \Big(time^{ub}(e) - blocked^{lb}(e) \cdot pn \Big) \cdot x_e.$$

Thus, during the construction of the abstract execution graph, we can directly calculate the edge weight $base^{ub}(e) := time^{ub}(e) - blocked^{lb}(e) \cdot pn$. Edges with different $base^{ub}$-weights can be merged to reduce the graph size without sacrificing precision in the overall compositional base bound calculation. On the downside, the calculated edge weights and thus the execution graph construction are no longer independent of the chosen penalty. Additionally, the abstract execution graph does not encode the full interference response curve.

5.6 Evaluation

In the previous sections, we have presented three approaches to establish timing compositionality. In the first approach, we adjust the processor hardware to ensure the existence of provable bounds on the effect of timing accidents. Here, we assess the impact of the required hardware changes on the actual performance.

The second approach requires sound penalties that incorporate all possible indirect effects caused by a timing accident. We employ the technique proposed in Section 5.4 to determine sound penalties *per benchmark*.

Last but not least, we evaluate our compositional base bound approach. We compare the precision of the obtained results w.r.t. the respective

interference response curves and examine the computational demand in terms of runtime and memory consumption.

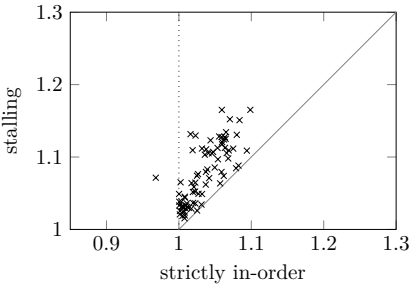
5.6.1 Hardware Design

In this section, we explore the performance impact of the two designs proposed in Section 5.3—the stalling mechanism and the strictly in-order pipeline—to establish compositionality. As a reference for both designs, we use our FPGA soft-core with its conventional in-order pipeline from Section 4.8.

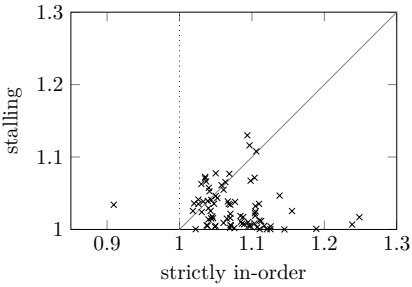
The only source of interference present in our Verilog prototype are additional cache misses, e.g. due to a preemption. The strictly in-order pipeline stalls instruction miss fetches when any data memory operation is pending—even if it results in a data cache hit later. Furthermore, the strictly in-order pipeline does not support branch prediction as it can lead to non-monotonic behaviour. The stalling approach requires the entire pipeline to stall upon each instruction and data miss in order to achieve compositionality.

To compare the performance overhead relative to our reference, a conventional in-order pipeline, we calculate the ratios of clock cycles needed to execute a given program on the modified processor relative to the reference. We obtain the clock cycles needed to execute a program using the same experimental setup as described in Section 4.8. We conduct the experiments for all our benchmarks, compiled with and without optimisations, as described in Appendix B. We provide results for different cache sizes and memory latencies. Figure 5.10 shows one scatter plot for selected configurations to compare the overhead introduced by the stalling mechanism and the strict memory access order. Each cross corresponds to one benchmark program. The x-coordinate (y-coordinate) of a cross represents the overhead introduced by the strictly in-order pipeline (stalling mechanism). A program whose cross is below (above) the diagonal line experiences less (more) overhead by the stalling approach than by the strict access order. In contrast to the stalling approach, the strictly in-order pipeline can perform better than the conventional in-order pipeline. In such a case, the cross is left of the dotted vertical line.

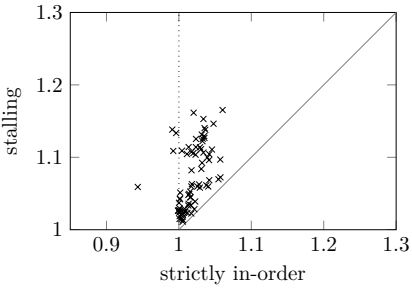
For all configurations, Figure 5.11 depicts the range and the geometric mean of the ratios over all benchmark programs. The average performance degradation caused by both hardware changes to establish compositionality



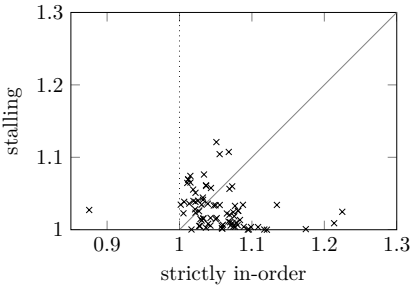
(a) Memory configuration (4, 64)@5.



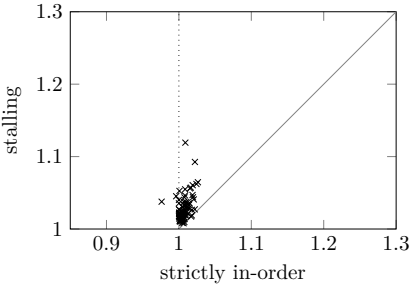
(d) Memory configuration (4, 64)@5.



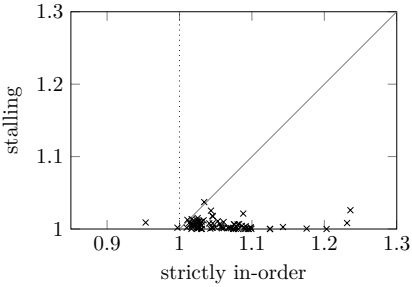
(b) Memory configuration (4, 64)@10.



(e) Memory configuration (4, 64)@10.



(c) Memory configuration (8, 256)@10.



(f) Memory configuration (8, 256)@10.

Figure 5.10: Scatter plot of the performance ratios relative to a standard in-order pipeline. Each cross corresponds to a benchmark program, compiled without (on the left) or with (on the right) optimisations. Different memory configurations ((cache line size in words, number of sets)@memory word latency).

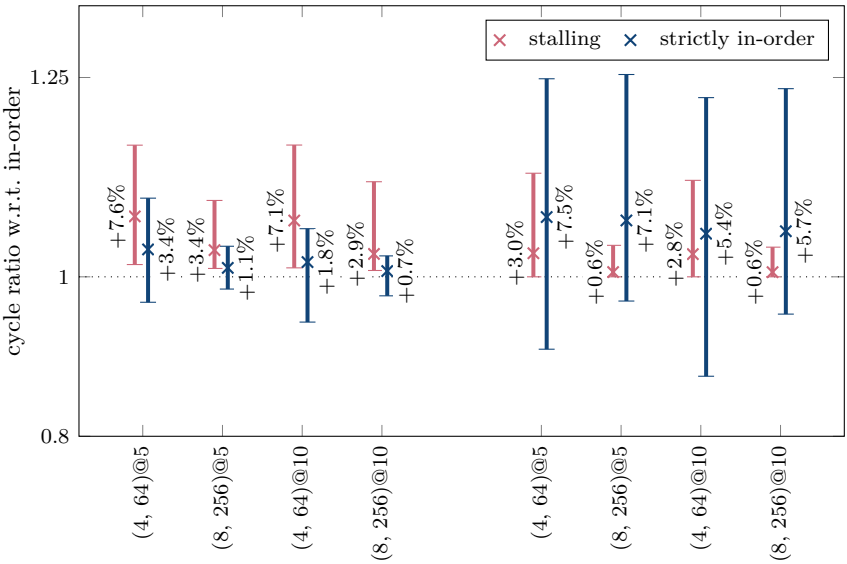


Figure 5.11: Performance in clock cycles relative to a standard in-order pipeline. Different memory configurations ((cache line size in words, number of sets)@memory word latency). On the left (right), minimum, maximum, and geometric mean over all our benchmarks without (with) compiler optimisations. Lower is better.

is well below 10% in terms of additional clock cycles required for execution. While the stalling approach performs better than the strictly in-order pipeline for optimised programs, the situation is reversed for non-optimised programs. This is explained by the different density of load/store instruction in the binaries: 28% for optimised binaries and 50% for non-optimised binaries. The stalling approach causes the pipeline to stall upon each data cache miss while the strictly in-order pipeline can hide a (small) part of the latency by overlapping execution of other instructions.

Larger instruction and data caches lead to lower cache miss rates. As a consequence, the stalling overhead upon cache misses reduces and the relative performance of the stalling approach improves. In a similar manner, the lower instruction cache miss rate improves the relative performance of the strictly in-order pipeline. Its relative performance, however, does not profit from a lower data cache miss rate as it does not reduce the overhead introduced by enforcing the strict order of memory accesses on the bus. A pending data memory access stalls an instruction miss fetch until the access reaches the memory pipeline stage even if it then turns out to hit the data cache. A lower data cache miss rate reduces the *absolute* number of clock cycles required by the strictly and the conventional in-order pipeline by roughly the same amount. Thus, the *relative* performance of the strictly in-order pipeline can even decrease slightly for a lower data cache miss rate.

A higher memory latency increases the impact of the memory on the overall performance and thus decreases the impact of differences in the pipeline design. Consequently, the ratio of clock cycles decreases with higher memory latencies, albeit only slightly. Last, we observe that the strictly in-order pipeline can indeed lead to shorter execution times compared to the conventional pipeline—in contrast to the stalling approach.

The performance degradation due to stalling or the enforcement of monotonic behaviour applied to a conventional in-order pipeline has been demonstrated to be in a small, acceptable range. However, this result *cannot* be transferred to more complex processors—e.g. with speculation and out-of-order execution—as they offer significantly more opportunities to hide the latency of timing accidents. Consequently, the impact of our hardware modifications on the performance of such processors will be significantly higher. We consider such an evaluation for a complex processor as important future work.

For interference on other resources which are not as tightly coupled with the pipeline as the caches—e.g. blocking at a shared bus—we expect the stalling approach to have a smaller impact on the overall performance. To observe a performance degradation, the processor pipeline requires significant capabilities to hide latency: not only the latency of the memory access itself but also the additional increase in latency needs to be hidden.

In this section, we have evaluated the actual performance impact of two hardware designs that support timing compositionality. We have not yet examined the benefits w.r.t. time and memory consumption of the timing analysis. In Section 5.6.3, we evaluate the compositional base bound approach—the only viable choice to establish compositionality by analysis for hardware platforms that do not support timing compositionality. There, we compare the resource consumption of the compositional base bound analysis and an analysis that can assume compositionality right away.

5.6.2 Sound Penalties

In this section, we want to estimate sound penalties required for compositional reasoning based on bounds that are valid under the absence of interference. To this end, we use the technique presented in Section 5.4 to compute per-benchmark penalties. A penalty that is sufficient to soundly cover the interference response curve of *any* program executed on the given hardware platform must be at least as high as these per-benchmark penalties.

As source of interference, we consider shared-bus blocking in a multi-core system. As a consequence, the latency of memory accesses performed by one core can be prolonged due to competing memory accesses generated by the concurrent cores.

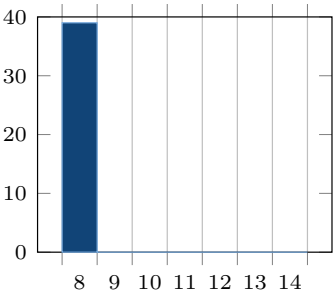
Experimental Setup We have implemented the analysis that is necessary to compute per-benchmark penalties as part of our low-level timing analysis framework LLVMTA introduced in Section 3.6 (Implementation/Tool Support). As the analysis is computationally expensive, we limit our experiments to the Mälardalen benchmark suite and our SCADE test cases as described in Appendix B. We conduct our experiments for the programs compiled with and without optimisations.

We compute the per-benchmark penalties on hardware configurations of varying complexity. We consider two different types of cores: the simpler type features a five-stage in-order pipeline, while the more complex one features an out-of-order pipeline with Tomasulo dynamic scheduling and speculative execution. Both pipeline types are described in more detail in Appendix A. Optionally, store instructions pass through a store buffer of size one that allows the pipeline to continue execution while the actual memory operation is performed in the background. The processors under analysis feature two cores.

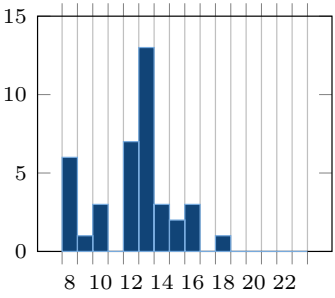
The cores access the shared bus via private instruction and data caches of size 1 KiB respectively. Each cache has a line size of 16 bytes and two cache ways that are managed by the LRU replacement policy. The shared bus employs round-robin event-driven arbitration and is directly connected to an SRAM background memory with either 5 or 10 cycles word latency.

Results For each of our benchmarks, we compute the minimal penalty needed to soundly approximate the respective interference response curve. We present the results in the form of histograms, i.e. we map each possible penalty to the number of benchmarks that require at least this penalty. For the benchmarks compiled without optimisations and executed on the in-order pipeline with varying parameters, we show the results in Figure 5.12. Figure 5.13 depicts the corresponding results for the out-of-order pipeline. Figures 5.14 and 5.15 show the histograms for the benchmarks compiled with optimisations.

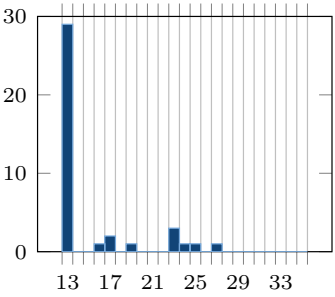
Figure 5.12a shows the results for the simplest hardware configuration with in-order pipeline and blocked stores, i.e. without store buffer. No benchmark experiences more than the worst-case direct effect penalty of eight clock cycles. The worst-case direct effect penalty is the time during which a load of an entire cache line can occupy the shared bus. It is composed of the memory latency—the time from the access request to the delivery of the first word of a cache line—plus one additional cycle per consecutive word loaded. Since store instructions occupy the memory stage of the pipeline until the access finishes, the in-order pipeline converges after a few cycles waiting for memory. Consequently, additional blocking on the shared bus has no impact on the pipeline state and thus cannot trigger any indirect effects.



(a) Blocked Stores, latency 5+3.

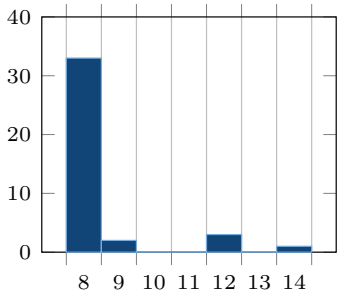


(b) Unblocked Stores, latency 5+3.

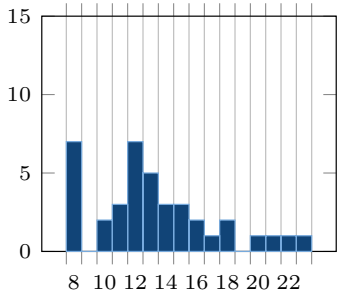


(c) Unblocked Stores, latency 10+3.

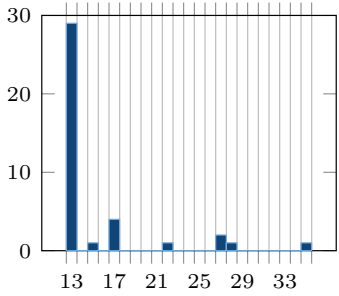
Figure 5.12: Histograms of the per-benchmark penalties required for a sound naive compositional analysis when the benchmark is executed on our in-order pipeline. Compiled without optimisations.



(a) Blocked Stores, latency 5+3.



(b) Unblocked Stores, latency 5+3.



(c) Unblocked Stores, latency 10+3.

Figure 5.13: Histograms of the per-benchmark penalties required for a sound naive compositional analysis when the benchmark is executed on our out-of-order pipeline. Compiled without optimisations.

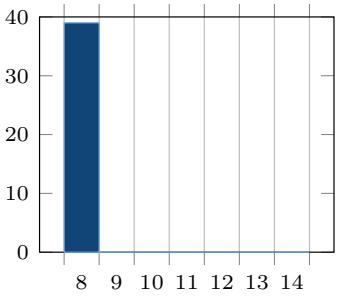
The out-of-order pipeline exhibits indirect effects for a few benchmarks despite the blocked stores (Figure 5.13a). Due to its superscalarity, the out-of-order pipeline can continue execution in its parallel functional units while waiting for memory. Such dynamically-scheduled pipelines are known to show (amplifying) timing anomalies [Lundqvist and Stenström, 1999].

Figure 5.12b depicts the results for the in-order pipeline with unblocked stores, i.e. with a single-entry store buffer, and a memory word latency of five clock cycles. A significant number of benchmarks exhibits indirect effects when executed on this hardware platform. A single benchmark, `compress` from the Mälardalen suite, even exhibits a penalty slightly more than *twice* as high as the direct effect penalty. The examination of the abstract execution graph reveals an anomaly similar to the one in Figure 5.2 presented in Section 5.1 (Validation of Compositionality Assumption). In contrast to the reordering of a *single* load with an instruction fetch, here, *two* memory operations are interchanged with the fetch of the consecutive instruction due to the latency prolongation of a preceding store.

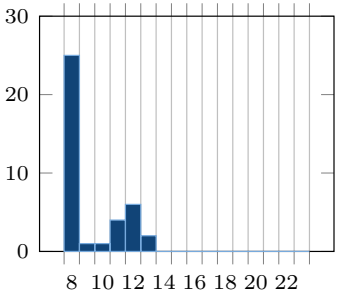
Figure 5.13b shows the corresponding results for the out-of-order pipeline. The number of test cases affected by indirect effects remains similar. As expected, the amount of indirect effects increases due to the higher complexity of the out-of-order pipeline. In the worst observed case, a sound penalty needs to be almost as high as *three times* the direct effect penalty.

The results for a higher memory word latency of ten clock cycles are depicted in Figures 5.12c and 5.13c. Compared to the lower memory word latency, fewer benchmarks are affected by indirect effects. For higher memory latencies, it is more likely that the processor pipeline converges before the pipeline could experience a latency prolongation due to shared-bus blocking. In these cases, additional blocking on the shared bus has no impact on the pipeline state and thus cannot trigger any indirect effects. Those benchmarks that are affected by indirect effects require a sound penalty of up to 2.1 times (in-order pipeline) or 2.7 times (out-of-order pipeline) the direct effect penalty.

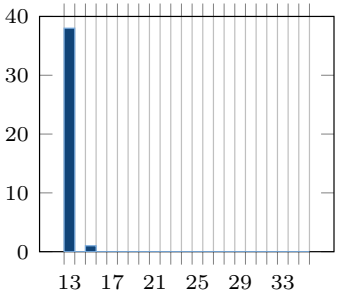
Figures 5.14 and 5.15 show the results when the benchmark programs are compiled with optimisations. Overall, we observe that fewer programs exhibit indirect effects compared to the results for the non-optimised programs. As explained in Appendix B, the compilation of programs with optimisations yields binaries that contain a smaller share of store instructions—on average 10% instead of 21% for non-optimised programs. When a store buffer is employed, the latency (prolongation) of store instructions can



(a) Blocked Stores, latency 5+3.

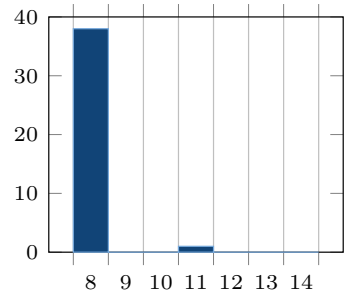


(b) Unblocked Stores, latency 5+3.

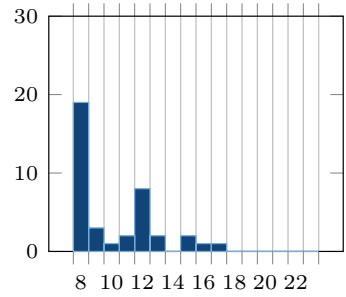


(c) Unblocked Stores, latency 10+3.

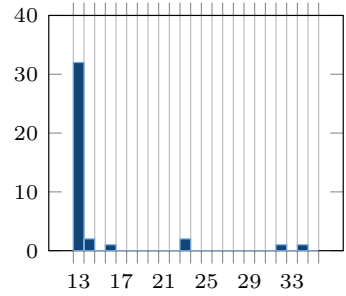
Figure 5.14: Histograms of the per-benchmark penalties required for a sound naive compositional analysis when the benchmark is executed on our in-order pipeline. Compiled with optimisations.



(a) Blocked Stores, latency 5+3.



(b) Unblocked Stores, latency 5+3.



(c) Unblocked Stores, latency 10+3.

Figure 5.15: Histograms of the per-benchmark penalties required for a sound naive compositional analysis when the benchmark is executed on our out-of-order pipeline. Compiled with optimisations.

usually be well overlapped with the execution of other instructions. As a consequence, the bus blocking of store instructions is often the source of amplifying timing anomalies. The smaller share of store instructions thus results in less potentially anomalous situations during execution and fewer affected programs.

However, the optimising compilation of programs can create code patterns that allow for amplifying timing anomalies even for very high memory latencies. Due to the lower number of load and store instructions, optimised programs exhibit longer sequences of instructions that do not access data memory. In particular within an out-of-order pipeline, the execution of a memory instruction can be overlapped with the execution of such a long chain of subsequent non-memory instructions—if data dependencies permit. As a consequence, the pipeline might not converge as fast as for non-optimised programs even for higher memory latencies. If the pipeline does not converge, there is potential for amplifying timing anomalies. During the inspection of our analysis results, we have found a code pattern with a store instruction in front of a loop without any memory instructions. Depending on the number of loop iterations, the execution of the store instruction could be overlapped with the execution of the loop for arbitrarily high memory latencies. If the code behind the loop follows a pattern similar to Figure 5.2, such a scenario could result in an amplifying timing anomaly for high memory latencies. This observation negates any intuitive reasoning to establish timing compositionality based on the assumption that the pipeline converges during a memory access, e.g. as found in [Wegener, 2017].

Conclusion In this section, we have evaluated a technique to derive a sound penalty for a given program, i.e. a penalty that accounts for any potential indirect effects. We have found that some programs exhibit amplifying timing anomalies that require a penalty of up to three times the direct effect penalty. The computation of a penalty that is sound for *any* program executed on a given microarchitecture is an open problem. Our experiments show, however, that such a general penalty needs to be at least three times higher than the direct effect penalty that is usually assumed in the literature. Depending on the amount of experienced interference, the usage of such a sound penalty will result in an intolerable amount of overall overestimation.

5.6.3 Compositional Base Bound

In this section, we evaluate the compositional base bound approach in terms of precision of the results and efficiency of the analysis. To this end, we compare the results obtained by the compositional base bound approach with the values of the interference response curve obtained by the analysis of [Jacobs et al., 2015]. In order to assess the efficiency of the compositional base bound computation, we compare its analysis runtime and memory consumption with the resource usage of the naive analysis that assumes the absence of interference. Note that the naive analysis *only* yields provably sound results for hardware platforms that support compositionality, e.g. those presented in Section 5.3.

As source of interference, we consider shared-bus blocking in a multi-core system. As a consequence, the latency of memory accesses performed by one core can be prolonged due to competing memory accesses generated by the concurrent cores.

Experimental Setup We have implemented the compositional base bound analysis in our low-level timing analysis framework LLVMTA introduced in Section 3.6 (Implementation/Tool Support). As benchmark programs, we use all the programs listed in Appendix B. We conduct our experiments for the programs compiled with and without optimisations separately.

We evaluate our compositional analysis approaches on hardware configurations of varying complexity. We consider two different types of cores: the simpler type features a five-stage in-order pipeline, while the more complex one features an out-of-order pipeline with Tomasulo dynamic scheduling and speculative execution. Both pipeline types are described in more detail in Appendix A. Optionally, store instructions pass through a store buffer of size one that allows the pipeline to continue execution while the actual memory operation is performed in the background. The processors under analysis feature either two, four, or eight cores.

The cores access the shared bus via private instruction and data caches of size 1 KiB respectively. Each cache has a line size of 16 bytes and two cache ways that are managed by the LRU replacement policy. The shared bus employs round-robin event-driven arbitration and is directly connected to an SRAM background memory with fixed latencies. We conduct experiments with different memory latencies.

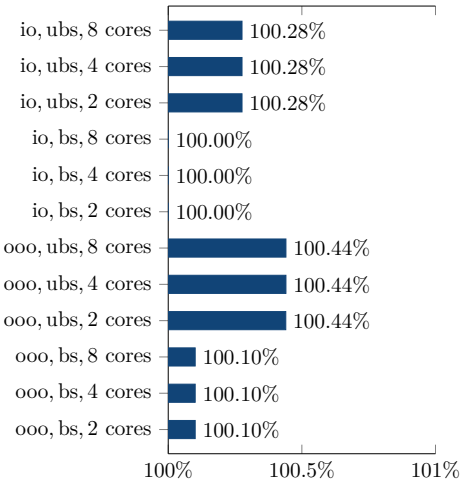
Results: Precision In order to assess the analysis precision, we first compare our sound compositional base bound approach with the unsound naive compositional approach. Additionally, we compare the results of the compositional base bound approach with the values of the interference response curve computed by the analysis of [Jacobs et al., 2015]. We choose the analysis of [Jacobs et al., 2015] as reference because it is sound—i.e. captures potential indirect effects—, reasonably precise—i.e. captures latency-hiding effects—, and computationally feasible in contrast to a more integrated and precise approach such as [Kelter and Marwedel, 2017].

As discussed at the end of Section 5.1, the shape of the interference response curves is usually concave. The compositional approaches approximate these curves by linear functions. The difference function of a linear and a concave curve is maximised at the boundaries of its domain. Since we employ event-driven round-robin bus arbitration, each program has a well-defined maximal amount of interference: each memory access can at most be blocked by one access per concurrent core. Consequently, we observe the largest difference in the computed bounds at zero or maximal interference.

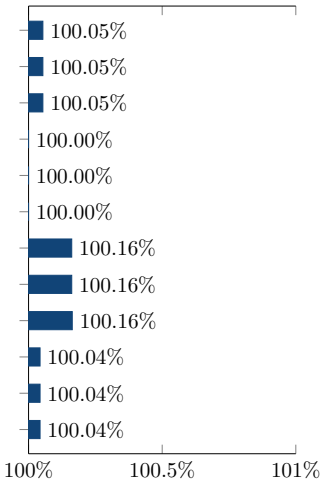
For each (hardware) configuration, we compute the geometric mean of the ratios of the compositional bounds to the interference response curve values over all benchmarks. To assess the variance of the per-benchmark results, we provide histograms of the ratios. For the sake of brevity, we show the histograms for only one configuration in Figure 5.17. We refer to Appendix C for the histograms of the remaining configurations.

Figure 5.16 and Subfigure 5.17a depict the results in case no interference occurs. Note that the results of the naive compositional analysis and the values of the interference response curves coincide at zero interference. Subfigure 5.16a illustrates the impact of a single-entry store buffer and a varying number of overall cores. Subfigure 5.16b illustrates the impact of different main memory latencies. While Subfigures 5.16a and 5.16b show the results for the benchmarks compiled without optimisations, Subfigures 5.16c and 5.16d show the results for the benchmarks compiled with optimisations.

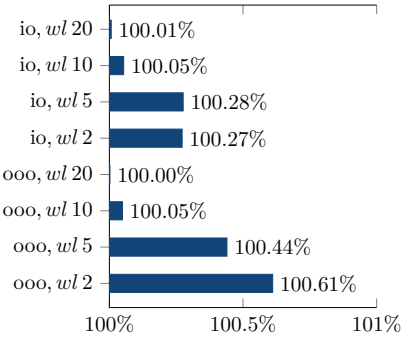
The ratios in Figure 5.16 reveal how much indirect effects a program experiences on average due to shared-bus interference. For all considered configurations, the amount of indirect effects experienced during a whole execution run is, on average, below one percent. Looking at the histograms at zero interference—in Subfigure 5.17a and Appendix C—no benchmark



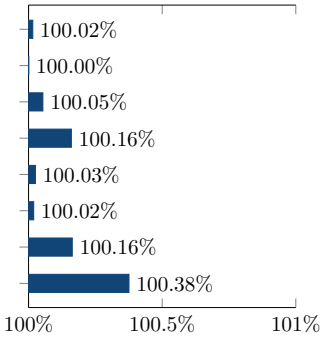
(a) Non-optimised programs. Memory word latency 5 processor cycles.



(c) Optimised programs.

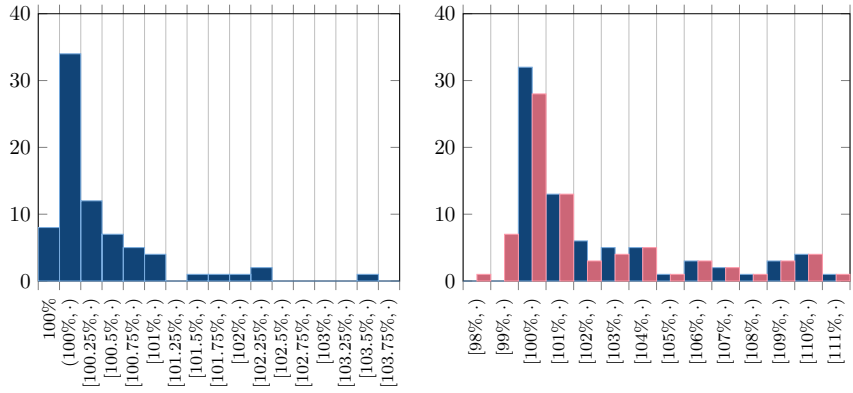


(b) Non-optimised programs. Unblocked Stores, two cores.



(d) Optimised programs.

Figure 5.16: Compositional base bound versus interference response curve at zero interference. In-order (io) or out-of-order (ooo) pipelined processor cores. Without (bs) or with single-entry store buffer (ubs). Varying number of cores and memory word latency (wl).



(a) At zero interference.

(b) At maximal interference.

Figure 5.17: Histogram of ratios of compositional base bound (left, blue bar) and naive compositional bound (right, red bar) to the interference response curve. Non-optimised programs executed on a dual-core machine with out-of-order pipeline, unblocked stores, and a memory word latency of 5.

exhibits indirect effects of more than four percent. This substantiates our intuition that it is more precise to account for the rare indirect effects in the base bound instead of the penalty.

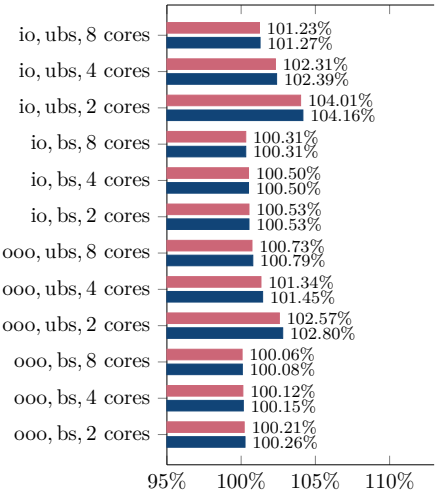
For the simplest hardware configuration "in-order pipeline, no store buffer (io, bs)", no indirect effects are encountered. Configurations featuring the out-of-order pipeline exhibit slightly higher indirect effects compared to the corresponding configurations with the in-order pipeline. An increase in the number of cores has almost no impact, i.e. additional cores and thus potentially longer bus blocking do not trigger more amplifying timing anomalies. Subfigure 5.16b shows the tendency of lower memory latencies to exhibit higher indirect effects. For lower memory latencies, it is more likely that the processor pipeline does not converge before it experiences the latency prolongation due to shared-bus blocking. The changes in the pipeline state while being blocked at the bus can then trigger amplifying anomalies. These observations are in accordance with our observations in Section 5.6.2.

The optimised programs tend to exhibit lower indirect effects on average as opposed to the programs compiled without optimisations. As can be seen in Appendix B, the programs compiled with optimisations contain a smaller share of store instructions. The latency (prolongation) of store instructions, however, can usually be well hidden when a store buffer is employed. The bus blocking of store instructions is thus often the source of amplifying timing anomalies.

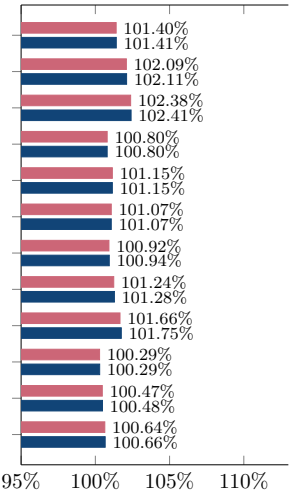
Figure 5.18 and Subfigure 5.17b depict the analysis results in case each program experiences its maximal interference on the shared bus. We consider a processor with n cores that are connected to memory via a shared bus with event-driven round-robin arbitration. Consequently, the maximal bus interference that a program p can experience is given by $m \cdot (n-1)$, where m denotes the number of memory accesses of p in the worst-case. The direct effect under maximal bus interference amounts to $m \cdot (n-1) \cdot ml$, where ml denotes the latency of the main memory. In a naive compositional analysis, the response time bound that corresponds to the maximal bus interference is computed as $\mathit{ideal} + m \cdot (n-1) \cdot ml$, where *ideal* is the worst-case execution time under the absence of interference. In the compositional base bound approach, *ideal* is replaced by the base bound that accounts for potential indirect effects due to interference. The respective bounds are normalised w.r.t. the maximal value of the interference response curve. The geometric means over all benchmarks are shown in Figure 5.18 that follows the same structure as Figure 5.16.

The results show that the differences between the naive compositional and the base bound approach are marginal on average. This means that accounting for potential indirect effects in the base bound has almost no effect on the precision of the results at maximal interference. However, it is strictly necessary to guarantee soundness: the naive compositional approach might underestimate the worst-case execution time in the presence of interference. Subfigure 5.17b reveals that the naive compositional approach underestimates the interference response curve at maximal interference for eight benchmarks by up to two percent. Though the results confirm that the compositional base bound approach introduces overestimation compared to the interference response curve—as the latency hiding effects are lost in any compositional analysis—it is still sufficiently precise.

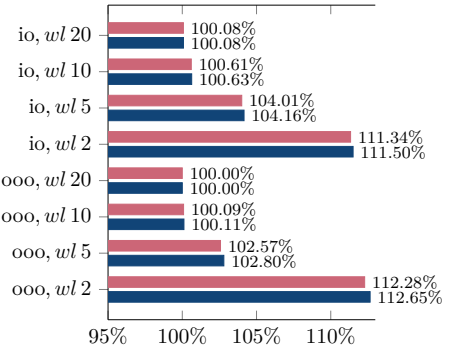
Furthermore, Subfigures 5.18b and 5.18d reveal that lower memory latencies allow for better hiding of the blocking effects. This is expected because



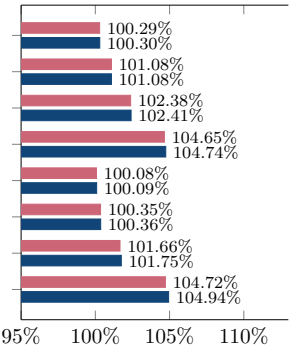
(a) Non-optimised programs. Memory word latency 5 processor cycles.



(c) Optimised programs.



(b) Non-optimised programs. Unblocked Stores, two cores.



(d) Optimised programs.

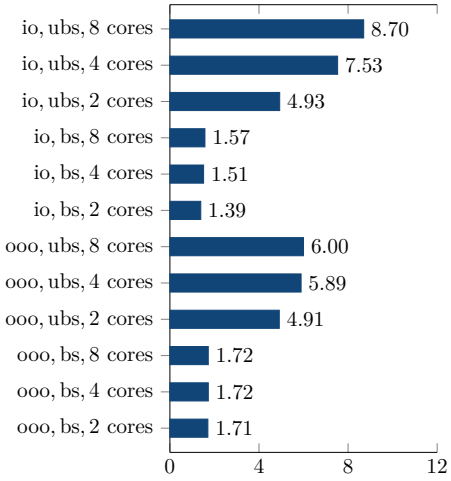
Figure 5.18: The naive compositional bound (upper, red bar) and the compositional base bound (lower, blue bar) versus interference response curve at maximal interference. In-order (io) or out-of-order (ooo) pipelined processor cores. Without (bs) or with single-entry store buffer (ubs). Varying number of cores and memory word latency (wl).

the longer a memory access takes on its own, the harder it is to hide any of the additional blocking. For the same reason, an increase in the number of processor cores reduces the relative amount of bus blocking hidden by overlapped execution.

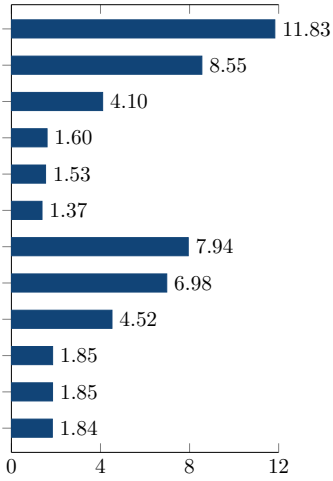
Maybe surprisingly, the configurations featuring an in-order pipeline show similar results w.r.t. hiding of additional latency as the configurations with an out-of-order pipeline. The main reason is that the modelled out-of-order pipeline as described in Appendix A is rather simple, e.g. it does not feature speculative memory accesses or multiple pending memory accesses. For a more complex pipeline, we expect better latency hiding capabilities and thus more overestimation introduced by compositional approaches compared to the interference response curve. Nevertheless, there are cases in which the in-order pipeline is indeed better at hiding latency than the out-of-order pipeline. Consider a store instruction followed by several independent arithmetic instructions. While the in-order pipeline has to execute the arithmetic instructions sequentially, the out-of-order pipeline can execute them in parallel due to its superscalarity. When the store experiences a prolongation due to bus blocking, the out-of-order pipeline might already be done executing the arithmetic instructions while the in-order pipeline can still overlap the waiting for memory with the execution of its remaining arithmetic instructions.

Compared to the benchmarks compiled without optimisations, the optimised benchmarks tend to exhibit a smaller or equal amount of latency hiding effects as can be seen in Figure 5.18. This is due to the lower share of store instructions among the overall number of memory accesses. The execution of store instructions can often be overlapped with the execution of the subsequent instructions if a store buffer is present. In case of load instructions, however, the subsequent instructions are usually data-dependent on the loaded value which hinders an overlapped execution.

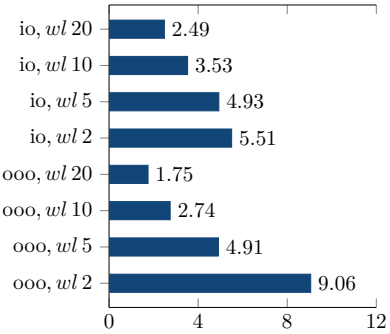
Results: Efficiency In this paragraph, we compare the resource consumption of the compositional base bound analysis with the consumption of the naive compositional analysis, i.e. a low-level analysis that assumes the absence of interference. To soundly account for any indirect effect, the compositional base bound analysis has to explore the effects of shared-bus interference on the microarchitectural behaviour of the given program. In our low-level analysis tool, we model shared-bus interference as a non-



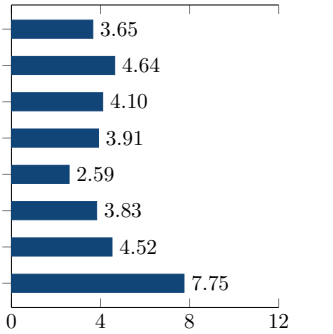
(a) Non-optimised programs. Memory word latency 5 processor cycles.



(c) Optimised programs.



(b) Non-optimised programs. Unblocked Stores, two cores.



(d) Optimised programs.

Figure 5.19: Compositional base bound analysis runtime versus naive compositional analysis runtime.

deterministic prolongation of each bus access. The exploration of this non-determinism is responsible for the increase in resource consumption of the compositional base bound analysis.

For hardware platforms that support compositionality, e.g. the strictly in-order pipeline, the naive compositional analysis yields already sound results. Thus, the results provided in this paragraph can also be interpreted as the savings in computational resources for the analysis when such platforms are used instead of conventional ones.

We have conducted all experiments on a desktop machine with an Intel® Core™ i7-7700 processor, clocked at 3.6GHz, and 64GB of main memory.

For each benchmark, we compute the ratio of resource consumption of the compositional base bound analysis w.r.t. the naive compositional analysis. Figure 5.19 shows the geometric mean of the analysis runtime ratios over all benchmarks for different configurations. In Appendix C, we additionally provide the sum of absolute analysis runtime over all benchmarks. Furthermore, we include the results for the main memory usage which show similar trends as the analysis runtime and are thus omitted here.

For the simpler configurations without a store buffer, the increase in resource consumption is relatively low. Recall from Section 3.6 that we can fast-forward the microarchitectural analysis to the end of an access if we detect that the pipeline state has converged. In these cases, the exploration of the effects of shared-bus interference is fast-forwarded and thus considered in an implicit manner. As a consequence, the compositional base bound analysis explicitly explores additional pipeline states only if the pipeline has *not* converged and thus might exhibit indirect effects.

As main memory latency increases, it is more likely that the pipeline converges before a memory access experiences a prolongation due to bus blocking. Due to the fast-forward technique, the analysis runtime ratio thus decreases for higher memory latencies.

Each additional concurrent core increases the potential prolongation of a blocked memory access and thus analysis runtime. However, with each additional core, it is more likely that the pipeline converges before it experiences the prolongation caused by the additional core. Taking our fast-forward technique into account, the analysis runtime ratio first increases and then levels off for higher numbers of concurrent cores.

The relative increase in analysis runtime for configurations with the out-of-order pipeline is similar to the relative increase for the corresponding

in-order pipeline configurations. The absolute numbers in Appendix C show that the analysis of the out-of-order pipeline is in general significantly more expensive.

The average increase in analysis runtime for the benchmarks compiled with optimisations is higher than for the non-optimised benchmarks for most hardware configurations. Optimised programs feature longer sequences of non-memory instructions compared to non-optimised programs. The execution of such an instruction sequence can be overlapped with the execution of a memory instruction within the pipeline. Consequently, upon a memory access, the pipeline takes longer to converge, which hinders our fast-forward technique.

Conclusion In this section, we have evaluated the compositional base bound approach. To the best of our knowledge, it is the only analysis approach that gives *sound* results for hardware platform that feature amplifying timing anomalies. We have demonstrated that the approach yields precise results compared to the naive compositional approach found in literature. The soundness comes at the price of increased analysis resource consumption because the analysis explores the (indirect) effects of interference on the execution time. However, the increase in resource consumption is usually well within a single order of magnitude. Consequently, we advocate the compositional base bound approach as the approach of choice to fulfil the compositionality assumption of state-of-the-art schedulability analyses.

Chapter 6

Related Work

In this chapter, we provide an overview of the literature related to the topics covered by this thesis. We structure the discussion of related work along the main chapters of this thesis.

6.1 Low-Level Analysis

A general introduction to low-level timing analysis can be found in the survey by Wilhelm et al. [2008]. The survey features a discussion of different methodologies to obtain timing bounds including static and measurement-based methods. The article concludes with an overview of the tools available at the time.

More recently, a trend towards probabilistic timing analysis and randomised hardware designs has evolved [Cazorla et al., 2013]. Such analyses provide a probability distribution of execution times either based on measurements and statistical methods or on a static model of the randomised hardware. The hope is that probabilistic timing analysis is more efficient and can derive high-confidence estimates that are tighter than the bounds provided by traditional timing analysis on deterministic hardware. To the best of our knowledge, it has neither been demonstrated how to actually build the required randomised hardware nor has it been shown that probabilistic timing analysis actually outperforms traditional timing analysis. In contrast, Reineke [2014] argues that randomised caches are not suitable for the use in hard real-time systems.

In this section, we limit ourselves to work related to static, non-probabilistic timing analysis. These techniques have been shown to provide reasonably tight bounds for realistic contemporary systems with acceptable analysis effort [Tan, 2006].

Control-Flow Reconstruction The compiler and linker influence the timing behaviour of a program, e.g. by optimisations and code/data placement in the address space. Consequently, low-level timing analysis takes a binary as input. The individual analyses are, however, based on control-flow graphs to represent programs. The first step in state-of-the-art analysers is the reconstruction of such control-flow graphs from the given binary.

Theiling [2002] describes a generic framework to reconstruct control-flow graphs from binaries. The framework comprises two parts. First, a classifier reads the raw bytes at a given address and decodes them into a machine instruction. Second, a bottom-up algorithm reconstructs the control-flow graph, starting from the entry-point address, using the classifier to correctly recognise instructions. To resolve switch tables and calls, the algorithm relies on knowledge about the used compiler and the patterns it generates.

Kinder et al. [2009] and Flexeder et al. [2010] propose static analyses based on abstract interpretation to reconstruct the control-flow graph. Their approaches do not rely on knowledge about the used compiler and can (partially) handle indirect jumps and calls whose target addresses are computed at runtime. To achieve this, they combine the control-flow reconstruction with a data-flow analysis.

In contrast to this general flow, our analysis tool (see Section 3.6) performs the analysis on the backend datastructures which LLVM provides through compilation. At this point, all compiler optimisations have happened and the machine program is ready to be passed to the linker. While this approach eliminates issues related to reconstruction, information about the actual address mapping is not directly available to the compiler.

Value and Control-Flow Analysis Prior to the hardware-specific analysis, value and control-flow analysis approximate the program semantics on the level of the instruction set architecture. Value analysis is used to bound the possible values of operands and the set of addresses possibly accessed by a memory operation. Control-flow analysis, in particular loop bound analysis, approximates the feasible paths through a program, e.g. how often a loop can execute.

The most common abstract domain for value analysis is the interval domain, where an interval $[l, u]$ describes all values between l and u. For each machine register and memory cell, an interval approximates the possible

values, respectively. This domain is discussed, among other value domains, by Cousot and Cousot [1977].

Miné [2006] extends the interval domain to octagons that describe the relation between two program variables x and y by constraints of the form $\pm x \leq c$ and $\pm x \pm y \leq c$, where c is a constant.

Granger [1991] introduces an abstract domain that captures linear congruences, i.e. constraints of the form $\sum_{i=1}^{n} a_i \cdot x_i \equiv c \mod m$, where x_i are integer-valued program variables and a_i, c, and m are integers. This representation subsumes simple congruences that constrain the possible values of a single variable. Such information is useful to describe the alignment of memory addresses.

Gustafsson et al. [2003] provide two approaches to bound the number of loop iterations from above. The first approach tries to match loops in the program to certain patterns such as a simple counting loop and to extract the loop bound from this pattern. If this approach fails, the second approach instruments the loop with a new loop counter variable and employs a value abstraction to derive loop bounds. In [Gustafsson et al., 2006], they extend their previous work to the detection of infeasible program paths, e.g. branches that are mutually exclusive.

Cullmann and Martin [2007] employ a three-step approach to loop bound analysis. First, an analysis identifies potential loop counter variables. Second, a data-flow analysis computes invariants on how these variables can change within a single iteration. Last, using their previous results and the actual loop condition, they calculate bounds on the number of loop iterations.

In [Ermedahl et al., 2007], the authors use value analysis to identify the number of possible states of all (integer-valued) variables that are involved in the loop condition. An additional invariant analysis is used to determine which of the variables can actually change during loop execution. Finally, assuming that all loops terminate, the number of different states of the non-constant variables provides an upper loop bound.

Microarchitectural Analysis The behaviour of caches has a major impact on the performance of a program. Consequently, there is a significant amount of work on caches in worst-case timing analysis. For reasons of brevity, we only provide an overview of the problems faced in cache analysis and refer to the recent cache survey by Lv et al. [2016] for details.

The field of cache analysis can be roughly divided into two lines of research. First, there are analyses that try to statically classify individual accesses as either cache hit, miss, or unknown, e.g. using under- and over-approximations of the cache contents [Alt et al., 1996]. Second, there are analyses that have a more global perspective and derive cumulative information such as "at most one of these accesses misses the cache". An example for the second type is persistence analysis such as presented in [Cullmann, 2013].

Different cache replacement policies pose different problems for cache analysis. An overview of the general predictability of cache replacement policies is given in [Reineke, 2009] while different abstract cache domains are described in [Grund, 2012]. Other common issues are imprecise address information for data cache analysis [Hahn and Grund, 2012], the analysis of multi-level caches [Sondag and Rajan, 2010], or the analysis of write-back caches [Blaß et al., 2017].

Besides cache analysis, the modelling of the pipeline in low-level analysis is important. In the following, we provide a brief overview of existing work on pipelines in low-level analysis.

In his dissertation, Engblom [2002] examines how to treat pipelines within low-level analysis. He presents a modular tool structure that is similar to the state-of-the-art tool architecture described in Section 3.3. Instead of performing a state-based microarchitectural analysis, he employs a simulator to estimate the timing of (sequences of) instructions under different so-called execution facts. He introduces execution facts to model all circumstances that influence the latency of instructions, e.g. the cache behaviour. The simulation of instruction sequences captures overlapping effects, but also anomalous behaviour that leads to an additional increase in execution time. His low-level analysis algorithm considers all sequences which can exhibit anomalous behaviour triggered by the first instruction in the sequence. He provides actual analyses for in-order pipelines while he classifies the analysis of out-of-order pipelines as too complex at that time.

Thesing has laid the foundation of today's state-of-the-art integrated pipeline and cache analysis in his dissertation [Thesing, 2004]. He introduces abstract pipeline states that explicitly maintain the control structure of concrete pipeline states and cut off the data-related parts which are approximated by the preceding value analysis. The abstract pipeline states are combined with the corresponding abstract cache states allowing for a precise analysis of this tightly-coupled system. The pipeline analysis determines

the set of reachable abstract cache/pipeline states for each basic block of the program. Uncertainty upon a cycle transition, caused by abstraction, is resolved by splitting the abstract state and following all possible cases. Consecutive splits lead to an exponential growth of the number of abstract states which results in a state space explosion, especially for complex systems. This analysis corresponds to our description of the microarchitectural analysis in Section 3.3. Thesing does not only provide the formal framework but also instantiates it for the analysis of two complex microprocessors, namely the Motorola ColdFire 5307 and the Motorola PowerPC 755.

Wilhelm [2012] tackles the state space explosion problem which results from analysis uncertainty in the explicit enumeration of abstract pipeline states. He uses techniques from symbolic model checking to represent sets of pipeline states as binary decision diagrams and to perform the analysis on this representation. The crucial requirement for this analysis technique is to keep the number of bits needed to encode the pipeline states low. He describes techniques to compress 32-bit addresses or larger hardware buffers into a small number of bits, e.g. by explicitly enumerating instructions and addresses relevant for the analysis of a single basic block. This enumeration is similar to our sketch in Section 4.2 of how to lift the progress-based abstract pipeline domain from a single instruction sequence to the microarchitectural analysis of whole control-flow graphs. Wilhelm also sketches an interaction between the symbolic pipeline analysis and the traditional cache analysis.

Heckmann et al. [2003] focus on the influence of the processor architecture on the precision and efficiency of the microarchitectural analysis. They discuss the influence of different cache replacement policies and show that the policies employed in commercial microprocessors do not necessarily allow for a precise analysis. Furthermore, they examine the interdependencies between cache, pipeline, and branch prediction. They experimentally determine the precision lost when the cache is modelled in isolation and the cache analysis needs to account for potential speculative accesses. They conclude that separate analyses are prohibitive and advocate for an integrated analysis as described in [Thesing, 2004].

In his dissertation, Maksoud [2015] evaluates the impact of the load-/store-buffer of the PowerPC 7448 on low-level analysis. He shows that smaller hardware buffers do not harm the worst-case timing predictions while significantly lowering the analysis time. Finally, he describes a compiler optimisation that introduces so-called sync instructions at specific pro-

gram points to normalise the abstract pipeline states and thus to help the microarchitectural analysis.

Path Analysis The early approaches to path analysis operate at the granularity of control-flow graphs of programs. The input to path analysis has been a control-flow graph annotated with a timing bound for each individual basic block. From the perspective of our low-level analysis in Section 3.3, this corresponds to a heavily compressed abstract execution graph.

Li and Malik have laid the foundation for state-of-the-art path analysis with their work on implicit path enumeration [Li and Malik, 1995]. The previous approaches to path analysis relied on an explicit enumeration of program paths which is very expensive. The key innovation is the encoding of the control-flow graph as a compact set of linear constraints that *implicitly* encodes all paths through the program. Furthermore, this encoding allows to easily add (linear) constraints to exclude infeasible paths, e.g. to bound the number of loop iterations. The calculation of an upper timing bound is then left to an off-the-shelf ILP solver.

Shortly after, Puschner and Schedl published their work on path analysis [Puschner and Schedl, 1997] which is very similar to [Li and Malik, 1995]. The authors take a more abstract perspective, viewing the path analysis problem as an instance of a maximum cost circulation in graphs. Besides the description of the ILP formulation, they observe that the encoding as integer linear program is only an approximation. The valuation returned by the ILP solver might not describe a connected path but unconnected path fragments. We have made similar observations, related to context-sensitive graphs and loop bound constraints, during the development of our low-level analysis tool described in Section 3.6.

In his dissertation, Theiling [2002] extends the implicit path enumeration introduced by Li and Malik to distinguish different execution contexts. Contexts are for example used to distinguish the behaviour of a callee for different function call sites. The microarchitectural analysis consequently determines a timing bound for each basic block in each possible context.

The weight maximisation within basic blocks (and context) prior to the path analysis leads to overestimation because it might introduce infeasible microarchitectural traces. Consider two consecutive basic blocks with their respective worst-case partial traces $\widehat{\tau}_1$ and $\widehat{\tau}_2$. If the microarchitectural state at the end of $\widehat{\tau}_1$ does not coincide with the start state in $\widehat{\tau}_2$, the addition of

their length exceeds the length of the worst-case trace through *both* basic blocks. To eliminate this overestimation, the idea of a state-sensitive graph as input to path analysis has evolved. This corresponds to the abstract execution graph in Section 3.3.

Matthies discusses a path analysis based on such abstract execution graphs, which he terms prediction files, in his diploma thesis [Matthies, 2006]. In contrast to previous ILP-based techniques, he proposes an algorithm based on depth-first search to compute upper timing bounds for the given acyclic graph. If the provided graph is cyclic, e.g. due to loops in the program, he applies his algorithm to the acyclic loop body first and then replaces the cycle by the calculated upper bound for the loop body. While this explicit approach offers better precision at higher analysis cost, a disadvantage is that additional constraints on infeasible traces cannot be added as easily and modularly as it is the case for an ILP formulation.

In his dissertation, Stein [2010] reconciles the implicit path enumeration with the more precise abstract execution graph. He tackles the problem of the larger input graph, compared to the previously used control-flow graphs, by applying a series of lossless and lossy compression steps. Next, he adapts the original encoding as integer linear program to the new input graph. Finally, he demonstrates how to incorporate constraints to eliminate infeasible microarchitectural behaviour using the example of cache persistence analysis.

6.2 High-Level Schedulability Analysis

Schedulability analysis determines for a certain scheduling policy whether a set of tasks is schedulable on a given hardware platform, i.e. whether all tasks meet their deadlines. Most schedulability analyses in literature follow the compositional approach. They use the results of preceding low-level analyses which characterise the execution behaviour of each task in isolation. Based on these characteristics, they incorporate additional schedule-dependent effects such as preemption delays or interference on shared resources. In this section, we only discuss related work that follows such a two-step approach. Other approaches that follow a more integrated view are discussed below in Section 6.5.

The foundation for contemporary real-time scheduling research has been laid by Liu and Layland [1973]. The authors discuss preemptive single-core

scheduling with fixed priorities (rate monotonic) as well as with dynamically adjusting priorities (earliest deadline first). Furthermore, they provide sufficient and necessary schedulability tests without the construction of an actual schedule.

This initial work has some simplifying assumptions such as neglectable preemption cost and context-independent task characteristics. For modern architectures with caches, dynamic random-access memory, or shared resources, these assumptions are either not true or introduce significant pessimism. Consequently, much research effort has gone into the elimination of these assumptions.

For reasons of brevity, we only discuss selected work related to the extension of the interface between low-level and schedulability analysis which affects the compositionality assumption. For a survey of multiprocessor real-time scheduling, we refer the interested reader to [Davis and Burns, 2011].

Atanassov and Puschner [2001] integrate the timing effect of DRAM refreshes into the response-time analysis for fixed-priority, preemptive scheduling. Refreshes are rare events that happen asynchronously to task execution and can prolong the affected memory accesses. A context-independent timing bound needs to consider refreshes for each memory access which introduces severe pessimism. The schedulability analysis can calculate the maximal number of refreshes during the worst-case response time of a task and account for their timing effect.

Schliecker and Ernst [2010] propose an analysis of multi-core systems with shared memory. The tasks, partitioned to individual cores, are preemptively scheduled with fixed priorities. First, the number of shared resource requests by each task within time intervals is approximated. Second, they derive the maximal amount of interference a task experiences given a certain arbiter, e.g. a work-conserving arbiter. Finally, the amount of interference is added to the computation time in isolation to obtain the worst-case response time. In a similar setting, Schranzhofer et al. [2011] present a different analysis of shared resource interference for adaptive arbiters. Their dynamic programming approach operates at the level of so-called superblocks that cannot be preempted and are characterised by maximal execution time and maximal number of accesses to the shared resource.

The cache-related cost incurred due to preemption has attracted quite some attention [Busquets-Mataix et al., 1996; Lee et al., 1996; Altmeyer et al., 2011; Lunniss et al., 2013]. The approaches use low-level characteristics of

the preempting and/or the preempted task to calculate the cache-related preemption delay within the respective schedulability analysis. While the authors of [Busquets-Mataix et al., 1996; Lee et al., 1996; Altmeyer et al., 2011] consider fixed-priority scheduling, Lunniss et al. [2013] examine scheduling with dynamic priorities according to earliest deadline first.

Recently, Altmeyer et al. [2015] have summarised the previous findings on DRAM refreshes, cache-related preemption delay, and shared resource interference in a generic compositional framework. We have presented their response-time analysis in more detail in Section 3.5 (Schedulability Analysis).

6.3 Progress-based Pipeline Abstraction

The major obstacle to efficient microarchitectural analysis is anomalous timing behaviour. Due to timing anomalies, a sound microarchitectural analysis needs to consider all possible successors upon a non-deterministic choice. We briefly review work on the definition and classification of timing anomalies as well as conditions that guarantee anomaly freedom. Last, we consider work towards more efficient microarchitectural analysis in the presence of anomalies.

Timing Anomalies In the common understanding, a timing anomaly describes a non-deterministic choice during analysis where the locally worst successor does not entail the global worst-case. This is in contrast to what we term an amplifying anomaly, i.e. a situation in which a local prolongation causes an even longer global prolongation. Some of the papers mentioned below treat both types of anomalies at once.

In 1969, Graham has already examined timing anomalies in the context of multi-processor systems. He has observed that reducing the execution time of tasks, relaxing dependencies, or adding additional cores can lead to a worse overall execution time. Most of his insights translate to the behaviour of modern dynamically scheduled out-of-order processors.

Initial work on timing anomalies in microarchitectural analysis has been conducted by Lundqvist and Stenström [1999]. They give definitions for anomalies, amplifying anomalies, as well as domino effects. A program modification technique is proposed to eliminate anomalies.

Besides the intuitive meaning of a *timing anomaly*, its formal definition has been an active research topic. Different definitions are given for example in [Wenzel et al., 2005; Reineke et al., 2006; Eisinger et al., 2006; Gebhard, 2010; Cassez et al., 2012]. The interpretation of the term "anomaly" thereby ranges from a purely hardware-based property ("consistently slower hardware" [Cassez et al., 2012]) to definitions based on analysis abstractions. Our definition given in Section 2.5 (Domination) belongs to the latter category. For reasons of brevity, we will not provide a comparison of the definitions.

Reineke et al. [2006] propose a definition based on transition systems and provide a classification into scheduling, speculation, and cache anomalies. Gebhard [2010] provides more examples including a domino effect triggered by MRU cache replacement and an anomaly caused by a mechanism of the in-order LEON2 processor to only partially fill lines in its LRU caches.

The automatic identification of anomalies or the automated proof of anomaly freedom has been a research objective for a long time. Eisinger et al. [2006] present a technique based on bounded model checking to automatically identify timing anomalies as counter examples to the query of anomaly freedom. The authors demonstrate their technique by identifying an anomaly on a simplified hardware model. For realistic systems, the proof of anomaly freedom or even the identification of an anomaly might be infeasible due to the computational complexity.

Reineke and Sen [2009] describe an approach to efficient low-level analysis in the presence of anomalies. They compute, for pairs of microarchitectural states, the maximal difference in execution time for any sequence of instructions. This difference can then be used to prune states during analysis. The maximal differences are computed by solving a recursive constraint system which encodes the possible state transitions. The approach might, however, not scale to the analysis of complex microprocessors.

Anomaly Freedom In [Engblom, 2002; Engblom and Jonsson, 2002], the authors model the pipeline behaviour for a sequence of instructions as an acyclic constraint graph. The graph has a node for each pair of instruction and pipeline stage and uses directed edges to describe dependencies, e.g. data dependencies and resource dependencies such as "ins_2 is fetched only after ins_1". As the authors state, this modelling is limited to simpler in-order pipelines. Out-of-order execution or a common memory bus as in the Princeton architecture will introduce cycles in the constraint graph. Note

that this cyclicity is closely related to the non-monotonicity of pipelines as we discussed in Chapter 4. Based on the above modelling, they derive interesting properties of pipelines. As an example, they show that any in-order pipeline is free of anomalies. Please note, that this statement does not conflict with our earlier observation in Chapter 4 as we consider a common memory bus. Even for an in-order pipeline, the common memory bus does, however, not perform all accesses in program order. If we model the common memory bus as pipeline stage in the constraint graph, we will thus obtain a cyclic constraint graph.

Similar to the above condition, Lundqvist and Stenström [1999] state that a processor with only in-order resources cannot exhibit any (amplifying) timing anomaly. In our work, we have discussed that this is not entirely true, even if the common memory bus behaves as an in-order resource. We have shown in Section 5.4 (Compositionality by Sound Penalty) that our strictly in-order pipeline design still exhibits (bounded) amplifying anomalies w.r.t. instruction cache misses.

Wenzel et al. [2005] introduce the notion of *resource allocation decision* as the principle behind timing anomalies. A resource allocation decision describes the possibility of different instruction orderings within a functional unit during execution, caused by a latency variation of an earlier instruction. The authors claim that the absence of such decisions implies that the processor does not feature timing anomalies. This is not true, unless the common memory bus is considered a functional unit [Hahn et al., 2015a]. Even in this case, the condition is not sufficient to eliminate amplifying anomalies.

Progress-based Pipeline Modelling In contrast to the enumeration of reachable pipeline states, Li et al. [2006] propose a new technique to deal with out-of-order processors in low-level timing analysis. The authors aim to compute for each instruction a timing interval in which the instruction enters or leaves the pipeline. This representation is closely related to our notion of progress of an instruction introduced in Chapter 4. Although not explicitly mentioned, their basic pipeline modelling, i.e. execution graph and dependence relation, coincides with the pipeline modelling of simpler in-order pipelines by Engblom [2002]. They extend the constraint graph proposed by Engblom with additional relations, called contention and parallelism relation, to model the behaviour of instructions in a superscalar

out-of-order pipeline. Using these additional relations, they prevent the problematic cycles in the dependence relation mentioned by Engblom. Based on the extended constraint graph, they present a fixed-point algorithm to calculate the timing intervals mentioned above. In contrast to state-of-the-art approaches, the algorithm neither enumerates the reachable pipeline states nor operates at the granularity of individual clock cycles. This promises higher efficiency but also lower precision than the state-of-the-art technique.

However, the approach has limitations as well. First, in out-of-order pipelines, the access sequences to the caches can vary due to reorderings and speculation in the pipeline. This becomes problematic if caches and pipelines are analysed separately. Second, to capture the overlapping between basic blocks, they incorporate so-called pro- and epilogues, i.e. instructions of the preceding and subsequent basic blocks that influence the execution of the current block. For complex out-of-order processors with several buffers, these pro- and epilogues might become prohibitively large.

Mohan and Mueller [2008a] introduce a hybrid approach to timing analysis that is claimed to handle even complex out-of-order processors. The approach requires processors with the ability to save and restore their physical implementation state, which they call snapshot. Based on this mechanism, they execute fragments of paths through the control-flow graph of a program from defined snapshots to obtain the fragments' timing. It remains unclear how to practically save and restore such snapshots on a real processor. Furthermore, an analysis that accounts for all possible machine states that arise during the execution will be expensive.

To improve efficiency, in [Mohan and Mueller, 2008b], the authors extend their approach by means to merge snapshots of machine states. As a motivating example, consider a program point where the control-flow of two paths A and B join and path B has taken more cycles to execute. An analysis of the subsequent instructions based on the snapshot at the end of B is not sufficient: path A, despite being shorter, might end in a snapshot that results in worse timing behaviour for the subsequent instructions. A sound, but expensive, analysis would need to consider such "anomalous" behaviour by exploring both snapshots. The authors propose to reduce the number of snapshots by merging, i.e. finding a snapshot that exhibits a timing behaviour that is at least as bad as the snapshots obtained at the end of A and B.

To construct such a snapshot, the authors take the respective maximum times at which an instruction can enter and leave a unit in the processors. This approach resembles the idea behind the progress-based join presented in Chapter 4 (Progress-based Abstraction). However, as they use the concrete cycle transformer implemented in silicon, recall that such a join is only sound if the machine behaves *monotonically* as discussed in Section 4.1 (Formalisation of Progress-based Abstraction). As discussed in Section 4.9 (Outlook: Monotonic Extensions), out-of-order pipelines—as considered by the authors—do not behave monotonically in general. Consequently, the merge operator as described in the paper cannot be sound. As an example, consider an instruction i after a control-flow join that needs a functional unit which is only available after the currently occupying instruction leaves the unit. Taking the maximum of the exit times, the merge rules out the case that i might actually start earlier on some execution path. But due to timing anomalies—similar to Figure 4.12 in Section 4.9—the case that i executes earlier can actually lead to the globally worst behaviour. Similarly, the authors make assumptions that are incorrect in the presence of timing anomalies, e.g. that a variable-latency instruction whose operands and thus latency are not known, always takes the maximum latency.

6.4 Compositionality

Most approaches to schedulability analysis, including the work discussed in Section 6.2, assume timing compositionality. While compositionality has been understood well intuitively—"in the sense that any shared resource delays are additive to the execution times" [Schliecker and Ernst, 2010]—there has been little research on the formal meaning or on how to achieve compositionality at all. Our work aims at bridging this gap by providing a formal definition in [Hahn et al., 2015b] and Section 2.6, and a comprehensive overview of techniques to achieve compositionality in Chapter 5.

The first definition of compositionality has been given by Wilhelm et al. [2009]. They classify an architecture as *fully compositional* if its abstract model does not exhibit timing anomalies, as *compositional with constant-bounded effects* if it does not exhibit domino effects, and as *non-compositional* otherwise.

The authors conclude that it suffices for an analysis of such a fully compositional architecture to only follow the local worst-case path. Although this

statement is true, it is in contrast to the intuitive idea of compositionality, namely to follow the local *best*-cases only and add timing penalties later on. At least, they mention that such an "even simpler timing analysis" [Wilhelm et al., 2009] is possible for the ARM7 that stalls upon each timing accident.

Their definition is based on "the" abstract model of an architecture, which is, however, not uniquely defined: there might exist different abstract models with and without timing anomalies. We base our definition on a given abstract model from the set of possible models. Consequently, we do not associate the compositionality property with an architecture itself.

An architecture whose abstract model exhibits domino effects is generally classified as non-compositional because an analysis has to follow all possible cases. In Section 5.5 (Compositional Base Bound), we have presented an analysis which follows all possible cases but still enables compositional schedulability analysis.

The definition of Wilhelm et al. [2009] is not sensitive to a chosen decomposition, i.e. it is not aware of the parts of the system which should be accounted for in a compositional manner. As a consequence, their requirement of complete anomaly freedom is overly restrictive. As an example, consider a model of an out-of-order pipeline which exhibits scheduling anomalies but stalls upon shared-bus blocking. The shared-bus interference can be handled in a compositional way, despite the timing anomalies.

Besides the issues we identified in the definition of [Wilhelm et al., 2009], the authors are right that timing anomalies, more precisely *amplifying* anomalies, seriously threaten *efficient* compositional analysis as we discuss in Chapter 5. Amplifying timing anomalies are known since the initial paper on timing anomalies in low-level analysis by Lundqvist and Stenström [1999]. But besides that, not much attention has been paid to amplifying anomalies in the past: they only prolong the local worst-case path and thus do not hinder a sound and efficient low-level analysis from ignoring the local best-case. Concerning compositionality, however, these anomalies turn out as the essential threat. Lundqvist and Stenström [1999] provide examples of amplifying timing anomalies triggered in dynamically-scheduled processors by uncertainty about the cache behaviour and the varying latencies for cache hits and misses. Such anomalies can also be triggered by uncertainty about the memory latency due to shared-bus blocking or DRAM refreshes. We show in Section 5.1 (Validation of Compositionality Assumption) that even a simple in-order pipeline with a small store buffer can cause amplifying anomalies.

In Section 5.4 (Compositionality by Sound Penalty), we state that there is no automated tool to calculate an upper bound on the maximal indirect effect caused by a timing accident. The work closest to such an automated tool is the technique of Reineke and Sen [2009] to calculate the maximal difference in execution time for a pair of microarchitectural states. However, the presented approach has not been demonstrated to scale to realistic architectures.

Whether a decomposition is successful in terms of increased efficiency and acceptable precision depends on how tightly-coupled the individual constituents are. The general decomposition of cache and pipeline behaviour leads to overestimation in the worst-case execution time without gaining much efficiency compared to an integrated approach [Heckmann et al., 2003; Faymonville, 2015]. The main reason for the observed imprecision is the fact that modern pipelines with speculation and out-of-order execution influence the cache access sequence. A sound cache analysis either needs to run a full-fledged pipeline analysis anyway or has to conservatively approximate any potential speculative and reordered accesses. For other cases, such as cache-related preemption delay or shared-bus blocking which depend on the actual schedule, a compositional approach will be the more viable choice.

6.5 Beyond Compositionality

Not all approaches to the analysis of multi-core or preemptively scheduled systems follow the common trend of assuming compositionality. In this section, we provide a brief overview of work beyond the compositionality trend. On the one hand, there are more integrated approaches that consider the effects of interference already during low-level analysis. On the other hand, there is the research area of composability and temporal isolation to mitigate interference by careful design of the system.

Integrated Approaches In his dissertation, Schneider [2003] proposes an approach to the analysis of preemptively scheduled systems with real-time operating system. The abstract model of the processor he investigates features domino effects caused by an additional cache miss. As a consequence, the block reload time cannot be bounded by a constant precluding a compositional treatment of the cache-related preemption delay. His approach partly integrates high-level schedulability analysis with low-level

WCET analysis. Scheduling information is fed to the low-level analysis to account for the additional cache misses due to preemption already in the computation time characteristics. Our compositional base bound approach can serve as compositional alternative for the analysis of such systems.

Dietrich et al. [2017] propose a response-time analysis for fixed-priority scheduled systems integrating the analysis of the real-time operating systems. Instead of bounding the execution time of each task, including the interrupt service routine and the operating system itself, in isolation, they examine the possible system-level contexts in which tasks are executed. Their work is based on a state transition graph that explicitly enumerates possible system states—including runnable threads, allocated resources, and active tasks—, and transitions between them. They also model the effect of asynchronous interrupt handling. Furthermore, they integrate cross-kernel flow facts excluding infeasible system-level paths to reduce pessimism. Although the approach does not inherently require timing compositionality, it might be necessary for efficiency reasons once complex hardware platforms, cache-related preemption delay, or interference on a multi-core are considered.

Kelter and Marwedel [2017] present their parallelism analysis to obtain precise timing bounds on multi-core systems. Their analysis explores all possible interleavings of co-running tasks on the multi-core. To limit the search space, a criterion is presented to identify infeasible interleavings. While low-level and high-level analysis are integrated, the authors make severe assumptions on the possible schedules: all tasks have the same period and each task is executed non-preemptively on a separate core. The evaluation shows benefits in precision, but also an increase in analysis runtime of two to three orders of magnitude, depending on the system under analysis.

If the state of the shared bus arbiter and the co-running tasks in a multi-core system are part of the abstract model, interference-induced timing anomalies show up. Shah et al. [2014] demonstrate by example that more interfering accesses of co-runners can influence the state of a round-robin arbiter such that it is more favourable for the timing of the remaining accesses of the considered task. If the arbiter state is not part of the abstract model, such as in [Jacobs et al., 2015], no timing anomalies occur. This substantiates the dependence on the chosen abstraction for the definition of an anomaly.

The multi-core analysis proposed by Jacobs et al. [2015] ranges between an integrated analysis in the sense of [Kelter and Marwedel, 2017] and a

low-level analysis assuming the absence of interference as performed in a compositional setting. The proposed microarchitectural analysis is core-modular, i.e. it only tracks the state of a single core, but incorporates the direct and indirect effects of interfering accesses. Each memory access can be non-deterministically blocked at the shared bus which allows a fine-grained analysis of the effect of interference on the core's state. Together with a core-modular arrival curve analysis that bounds the amount of interference per time interval, the authors calculate co-runner-sensitive worst-case response times. The evaluation is performed for a system with event-driven round-robin bus arbitration. In this thesis and in [Hahn et al., 2016], we use this interference-sensitive analysis to sample interference response curves (Section 5.1 (Validation of Compositionality Assumption)) and to compute compositional base bounds (Section 5.5 (Compositional Base Bound)).

Composability and Temporal Isolation Composability allows to combine different components without changing the behaviour of the components. As an example, if a set of tasks is integrated on a single, composable multi-core platform, the timing behaviour of the individual tasks should not be influenced by the co-running tasks. In other words, the tasks are *temporally isolated*. In such a system, the traditional analysis methods known from single-core verification can be reused.

Designing a timing-composable system is difficult as each component has to support temporal isolation. Akesson et al. [2010] distinguish the concepts of composability and predictability. They discuss how to achieve both properties at the level of processor cores, the interconnect, and the background memory. Bui et al. [2011] survey the components that have to provide temporal isolation including the pipeline, the local memories, the interconnect, and the DRAM controller. Goossens et al. [2013] survey their previous work on the CompSOC architecture that provides temporal isolation between applications by, e.g., employing time-division multiplexing techniques.

Time-division multiple-access (TDMA) techniques are popular to achieve access to a shared resource in a temporally isolated fashion. Kelter et al. [2011] propose a low-level analysis to track the current offset within a TDMA cycle to obtain a bus-aware worst-case execution time bound.

A technique to achieve temporal isolation of state-based resources such as caches is *partitioning*. Cache partitioning can either be realised in hardware

or in software, i.e. by an operating system. A hardware-based solution is to partition the set of cache ways as described as column caching by Chiou et al. [2000]. Taylor et al. [1990] mention a software-based technique called page colouring which has later been used to partition shared caches [Lin et al., 2008]. Plazar et al. [2009] employ instruction cache partitioning to tighten timing bounds in a multi-task setting. The partitioning of shared caches is commonly used to achieve temporal isolation in multi-core systems. Guan et al. [2009] extend their task model by cache capacity demands and adjust the schedulability analysis to ensure that the capacity demands are always met.

6.6 Hardware Design for Predictability

General-purpose hardware design aims at high average-case performance. When it comes to timing-critical systems where upper timing bounds matter, such designs cause issues in the timing verification process. General-purpose hardware is considered to be unpredictable w.r.t. timing performance due to high timing variability and a strong history-dependence. Additionally, timing anomalies and domino effects hinder efficient analyses. The ultimate goal is to design a processor that is predictable and efficiently analysable while showing good performance. Our work contributes the strictly in-order pipeline design which supports compositionality and anomaly freedom without sacrificing too much performance. In the following, we survey related work in the field of predictable hardware design.

In the scope of the PREDATOR project, Wilhelm et al. [2009] give advice concerning the design of future architectures for the time-critical domain. Their work covers pipelines, buses, and memory hierarchies with a strong focus on caches and their analysis. They recommend to use predictable pipelines such as the one found in the simple ARM7 processor which they conjecture to be compositional.

The MERASA project [Ungerer et al., 2010] targets hardware design and analysis tools for multi-core embedded processors in a mixed-criticality setting. They implement temporal isolation for hard real-time tasks against other tasks. The proposed processor cores feature simultaneous multi-threading that prioritises the hard real-time thread. As core-local memories, they use scratchpads for the hard-real time threads and caches for the remaining threads. The shared bus with bounded waiting time connects the

cores to a dynamically partitioned cache. The DRAM device is accessed through a closed-page, predictable memory controller with round-robin arbitration. Their low-level analysis calculates context-independent characteristics, i.e. the low-level analysis accounts for the maximal interference effects.

The aim of a precision-timed (PRET) machine is repeatable timing, i.e. a program always takes the same execution time for a given input. Liu et al. [2012] describe an implementation of such a PRET machine whose timing is repeatable and predictable with competitive performance. They employ a five-stage, thread-interleaved pipeline with four threads, local scratchpad memories, and the repeatable DRAM controller of Reineke et al. [2011]. The interleaved execution of hardware threads eliminates hazards and requires no forwarding circuits. While the performance w.r.t. throughput is competitive, the single-thread performance is significantly decreased as each thread is only executed every fourth cycle. The FlexPret extension [Zimmer et al., 2014] introduces flexibility to the hardware thread schedule. The single-thread performance of a thread can be increased by scheduling the thread more often. FlexPret targets at mixed-criticality systems and consequently supports hard and soft real-time threads.

De Dinechin et al. [2014] present the Kalray MPPA®-256 many-core processor that suits time-critical applications. The many-core is divided into 16 clusters that are connected via a Network-on-Chip. Each cluster features 16 cores with a seven-stage, statically scheduled pipeline that can issue up to five instructions per cycle. Each core has separate instruction and data caches following the LRU replacement policy, and a small write buffer. The authors state without further explanation that their core qualifies as fully timing compositional in the sense of [Wilhelm et al., 2009], i.e. it does not exhibit timing anomalies.

Recently, within the T-Crest project, Schoeberl et al. [2015] have proposed a design for a predictable multi-core platform. A TDMA-based Network-on-Chip provides core-to-core communication while a memory tree provides access to a single DRAM controller for all cores. Each core, called Patmos, features a dual-issue, statically scheduled pipeline and a variety of local memories such as method, stack, and data cache as well as an optional scratchpad. In contrast to ordinary processors, all caches are filled in the memory pipeline stage only: upon a call or return instruction in the memory stage, the method cache is filled with the next function. Although the authors claim in [Schoeberl et al., 2014] that Patmos supports

compositionality and is free of anomalies, to the best of our knowledge, there is neither a formal proof nor an (intuitive) argument to support that claim.

Chapter 7

Conclusions and Future Work

In hard real-time systems, it is crucial to verify that the system meets its timing requirements under all circumstances. To this end, timing analysis computes the worst-case response time of each task in the system. To ensure soundness, timing analysis accounts for the influence of the program inputs, the microarchitectural state, and the concurrent tasks competing for shared resources.

Modern microarchitectures commonly exhibit counter-intuitive timing behaviour that complicates the timing analysis or renders it less efficient. *Timing anomalies* prevent the (low-level) analysis from pruning parts of the search space because a locally fast execution might result in a globally slow execution. *Indirect effects* hinder a compositional (high-level) analysis since a locally slowed-down execution can lead to an even slower global execution. How to deal with such counter-intuitive behaviour is thus a key concern of timing analysis.

Contributions

In this thesis, we have made the following contributions to state-of-the-art timing analysis.

First, we have introduced the idea of abstracting pipeline states based on the *progress* of a program's execution within the pipeline. An abstract pipeline state $\widehat{p}$ thereby compactly represents all concrete pipeline states with at least the progress of $\widehat{p}$. We have presented a generalised formalism that allows to reason about the soundness of progress-based abstractions. Defining the abstract cycle behaviour of a given microarchitecture is difficult in general. However, if a microarchitecture behaves monotonically w.r.t.

progress, its concrete cycle behaviour can be used as the abstract cycle behaviour.

Second, we have presented the *strictly in-order pipeline* that behaves monotonically w.r.t. the progress of a program's execution. In contrast to a conventional in-order pipeline, it enforces that all operations on the memory bus, i.e. instruction and data accesses, occur in program order. The monotonicity of the cycle behaviour enables us to prove the absence of timing anomalies which in turn enables a more efficient timing analysis.

Third, we have examined the compositionality assumption that is crucial for the soundness of many approaches to high-level analysis proposed in the literature. An experimental validation has revealed that even simple microarchitectures exhibit indirect effects that have previously been ignored. As a consequence, we have introduced the *compositional base bound* analysis that accounts for potential indirect effects already during the low-level analysis. While the compositional base bound analysis is applicable to any microarchitecture, the analysis demands more computational resources. Last but not least, we have shown how to adjust microarchitectures in order to achieve timing compositionality on the hardware level.

Conclusions

In conclusion, a *sound* compositional timing analysis of modern real-time systems that feature multi-core processors and preemptive execution is possible when indirect effects are considered. For the first time, our *compositional base bound* analysis enables the sound application of existing high-level schedulability analyses to realistic and contemporary hardware platforms.

On the downside, timing analysis demands an immense amount of computational resources to account for all anomalies and indirect effects present in contemporary microarchitectures. We advocate that the (provable) predictability of a microarchitecture's timing behaviour should be a major design goal for future processors used in a hard real-time context. The strictly in-order pipeline that behaves monotonically w.r.t. execution progress is a first step in this direction.

Future Work

The strictly in-order pipeline has been designed with monotonicity in mind to ensure timing predictability, i.e. timing compositionality and the absence of timing anomalies. It is future work to review performance-enhancing features other than pipelining in order to decide to what extent they can be employed in a processor with monotonic timing behaviour—if at all possible.

Since the (timing) behaviour of processors is generally complex and often subtle, it is important to have formal proofs of properties such as monotonicity. To this end, it is desirable to carry out the proofs in an automatic or interactive theorem prover.

A non-relational progress abstraction is sufficient to analyse pipelines with a monotonic timing behaviour. However, for non-monotonic pipelines, e.g. a conventional in-order pipeline, no sound abstract cycle transformer could be found that is guaranteed to make progress. A *relational* abstract domain that additionally tracks the combined progress of instruction pairs might be powerful enough to enable such a sound abstract transformer.

Appendix A

Computer Architecture: Concepts

A.1 In-Order Pipeline

Pipelining is a technique to increase the instruction throughput of a processor, i.e. to increase the number of completed instructions per clock cycle. The fundamental observation is that during execution, each instruction performs roughly the same operations where each operation requires only a subset of the processor's logic. Consequently, consecutive instructions can perform distinct operations *in parallel* to better utilise the overall processor. In accordance with these operations, the processor's logic is grouped into consecutive *pipeline stages* of similar length where each stage performs one operation.

A common separation of an instruction's execution into five operations is the following [Hennessy and Patterson, 2012]:

(i) *fetch* the instruction from the instruction memory to the processor core,
(ii) *decode* the instruction and fetch the required register operands,
(iii) *execute* the instruction using the respective operand values,
(iv) optionally perform a data *memory* access, and
(v) *write back* the respective result into a register.

In an *in-order* pipeline, each stage processes the instructions in the order given by the binary program. This is in contrast to an out-of-order pipeline as described in Appendix A.3 that dynamically reorders the execution of instructions. In Figure A.1, we depict a conventional in-order pipeline with five stages, described in more detail e.g. in [Hennessy and Patterson, 2012].

In the ideal case, the instruction throughput of an n-stage in-order pipeline increases by a factor n w.r.t. a non-pipelined machine. Note that the latency of a single instruction is not reduced by pipelining. There are

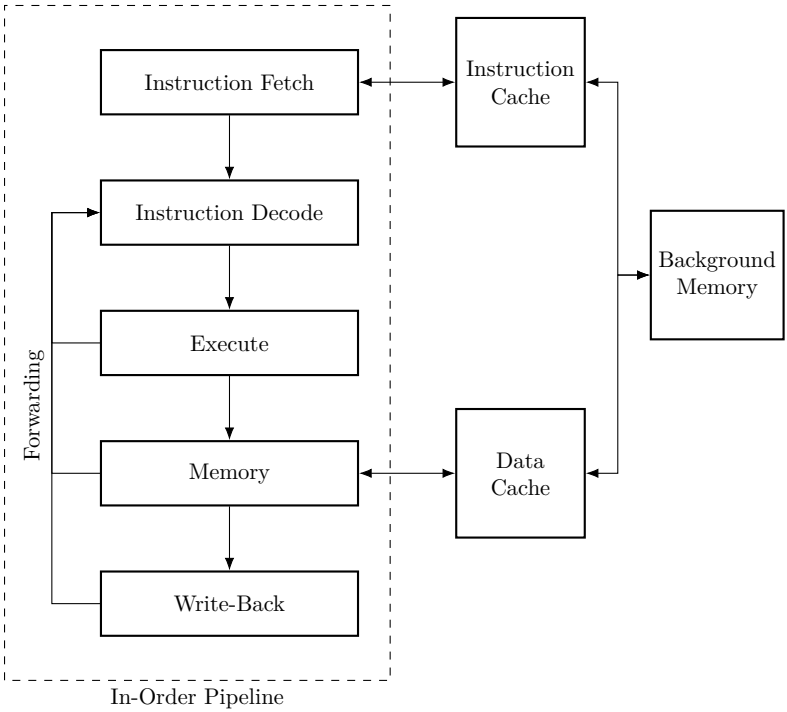


Figure A.1: Schematic view of a five-stage in-order pipeline with memory hierarchy.

multiple reasons why this theoretical speed-up is never reached. First, not each instruction requires *all* operations, e.g. an addition performs no data memory access and a branch does not write-back a result. Second, the circuit delays of the individual stages differ. The maximal clock frequency of the pipelined machine is determined by the slowest stage. Third and most importantly, the operations of the different instructions within the pipeline are often *not independent* which hinders their parallel execution. Such cases are called pipeline *hazards*.

We distinguish three types of hazards. A data hazard occurs if an instruction in the decode stage needs to read a register that will be written by previous instructions still in the pipeline. Certain data hazards can be

resolved by a technique called forwarding that forwards the required value from the respective pipeline stage as soon as it is ready. If the register of interest is written by a preceding load instruction, the decode stage is stalled and has to wait until the data memory access has been finished. A control hazard arises if the outcome of a branch, and thus which instruction to fetch next, is not known yet. Instead of waiting for the branch to resolve, the pipeline can speculatively fetch the next instructions. If the speculation turns out to be wrong, the speculatively fetched instructions are rolled back and the pipeline restarts at the correct program location. Last, structural hazards describe scenarios in which different instructions want to access the same resource at the same time. As an example, consider a memory load and an instruction fetch that both miss their respective cache and thus need to access the background main memory. The pipeline grants access to one stage and stalls all other requesting stages.

In the evaluation in Sections 4.8 and 5.6, we use a variant of the described five-stage in-order pipeline. First, it features forwarding to reduce the number of data hazards. Second, its fetch stage can detect and execute unconditional branches right away which reduces the number of control hazards. Furthermore, to limit the impact of control hazards caused by conditional branches, our pipeline features a simple branch speculation. Based on the branch offset encoded in the branch instruction, it predicts backward branches as taken and forward branches as not-taken. The rationale behind this scheme is that branches with a backward heading offset are usually used to implement the condition of a loop.

Formalisation of Cycle Behaviour In this paragraph, we formally define the behaviour of our five-stage in-order pipeline. In accordance with Section 4.1 (Formalisation of Progress-based Abstraction), we provide the pipeline cycle behaviour for a fixed sequence of instruction instances which represents the instructions fetched during a specific program execution. $\mathcal{I}_d$ denotes the set of dynamic instruction instances from this sequence. We order the instruction instances according to their position within the sequence, i.e. $ins_n < ins_m$ denotes that instruction instance ins_n occurs before ins_m. With each dynamic instruction $ins \in \mathcal{I}_d$, we associate its operation code $opc(ins)$, its operand registers $ops(ins)$, and its destination registers $target(ins)$.

To model speculative fetching, the set $\mathcal{I}_d$ includes instruction instances

$$
\begin{aligned}
evs := \{&(\text{dcmiss}, i, p(i), p'(i)) \mid p'(i) \sqsubset (EX, 0) = p(i) \wedge \neg dchit(i) \wedge opc(i) = load\} \\
\cup \{&(\text{dchit}, i, p(i), p'(i)) \mid p'(i) \sqsubset (EX, 0) = p(i) \wedge dchit(i) \wedge opc(i) = load\} \\
\cup \{&(\text{icmiss}, i, p(i), p'(i)) \mid p'(i) \sqsubset (pre, 0) = p(i) \wedge \neg ichit(i)\} \\
\cup \{&(\text{ichit}, i, p(i), p'(i)) \mid p'(i) \sqsubset (pre, 0) = p(i) \wedge ichit(i)\}
\end{aligned}
$$

Figure A.2: Definition of the events that occur during a cycle transition of a five-stage in-order pipeline.

that are fetched speculatively but never executed. With each branch instruction, we associate whether the subsequent speculation will result in a flush (*missspec*) for the specific program execution and which instructions have been potentially fetched speculatively while resolving the branch condition (*specfetch*).

Parts that do not belong to the pipeline control are modelled by external functions. The function *ichit* (*dchit*) returns true if and only if the fetch (data) access of an instruction instance hits the instruction (data) cache. *exlat* returns the execution latency of an instruction instance whose latency might depend on the operand values, in particular for floating-point instructions. $memlat_f$ ($memlat_d$) returns the memory latency that the fetch (data) access of an instruction instance experiences. If abstraction is employed for the non-pipelined part, e.g. a cache abstraction, the external functions might provide uncertain answers. In this case, the cycle behaviour follows all successor configurations permitted by the uncertain external information.

A pipeline state $p \in \mathcal{I}_d \to \mathcal{S} \times \mathbb{N}_0$ maps each instruction to its current pipeline stage and the number of remaining cycles to finish the current stage. The state can also be expressed as a pair of functions

$$p = (stage, cnt) \in (\mathcal{I}_d \to \mathcal{S}) \times (\mathcal{I}_d \to \mathbb{N}_0)\,.$$

We formally define the cycle behaviour $cycle(p)(evs)(p')$ of an in-order pipeline by the equations in Figures A.2 and A.3. The primed variables correspond to the pipeline configuration p' after the cycle transition. $\sqsubseteq$ corresponds to the progress order described in Section 4.3 (Progress of In-Order Pipeline). Variables i and j denote instruction instances from $\mathcal{I}_d$.

$$p' := \lambda i \in \mathcal{I}_d.\ \begin{cases} (stage'(i), latency(i)) & : ready(i) \wedge willbefree(stage'(i)) \\ (stage(i), cnt'(i)) & : \text{otherwise} \end{cases}$$

$$cnt'(i) := \begin{cases} cnt(i) - 1 & : cnt(i) > 0 \\ 0 & : cnt(i) = 0 \end{cases}$$

$$stage'(i) := \begin{cases} post & : stage(i) = WB \vee flush(i) \\ WB & : stage(i) = MEM \\ MEM & : stage(i) = EX \\ EX & : stage(i) = ID \\ ID & : stage(i) = IF \\ IF & : stage(i) = pre \end{cases}$$

$$\begin{aligned} ready(i) &:= cnt(i) = 0 \\ &\wedge ((stage(i) = EX \wedge opc(i) = store) \Rightarrow busfree) \\ &\wedge ((stage(i) = EX \wedge opc(i) = load) \Rightarrow dchit(i) \vee busfree) \\ &\wedge ((stage(i) = EX \wedge opc(i) = branch \wedge missspec(i)) \\ &\qquad \Rightarrow \neg\exists j.\ j \in specfetch(i) \wedge stage(j) = IF \wedge cnt(j) \neq 0) \\ &\wedge (stage(i) = ID \Rightarrow \neg ophaz(i)) \\ &\wedge (stage(i) = pre \Rightarrow next(i) \wedge (ichit(i) \vee ibusfree)) \end{aligned}$$

$$\begin{aligned} willbefree(s) &:= s = post \\ &\vee (\neg\exists i.\ stage(i) = s) \\ &\vee (\exists i.\ stage(i) = s \wedge ready(i) \wedge willbefree(stage'(i))) \end{aligned}$$

$$busfree := \neg\exists i.\ (stage(i) = IF \vee stage(i) = MEM) \wedge cnt(i) > 0$$

$$ibusfree := busfree \wedge \neg\exists i.\ opc(i) \in \{load, store\} \wedge p(i) = (EX, 0) \wedge \neg dchit(i)$$

$$latency(i) := \begin{cases} memlat_I(i) & : stage'(i) = IF \wedge \neg ichit(i) \\ memlat_d(i) & : stage'(i) = MEM \wedge \neg dchit(i) \\ exlat(i) & : stage'(i) = EX \\ 0 & : \text{otherwise} \end{cases}$$

$$\begin{aligned} next(i) &:= stage(i) = pre \wedge \forall j < i.\ stage(j) \neq pre \\ flush(i) &:= \exists j.\ stage(j) = EX \wedge cnt(j) = 0 \wedge j = branch \\ &\wedge misspec(j) \wedge i \in specfetch(j) \wedge (stage(i) = IF \Rightarrow cnt(i) = 0) \\ ophaz(i) &:= \exists o \in ops(i)\, \exists j.\ p(j) \sqsupset (MEM, 0) \wedge opc(j) = load \wedge o \in target(j) \end{aligned}$$

Figure A.3: Equations expressing the cycle behaviour of a five-stage in-order pipeline.

A.2 Memory Hierarchy

In an ideal world, a processor connects to an inexpensive, large, and fast memory that stores code and data. However, there is no single type of memory that has these properties. Rather, there is a multitude of different semiconductor technologies to implement memory, each with its specific advantages and drawbacks. To create the illusion of a large and fast memory at an acceptable cost, a hierarchy of memories is employed [Hennessy and Patterson, 2012]. A large but slow main memory can store a large amount of data while fast but small memories provide a fast access path to the processor by buffering the most frequently used data. In the following, we review some of the major components of the memory hierarchy as found in contemporary systems.

A.2.1 Main Memory

There are different semiconductor technologies to manufacture memory with specific characteristics. For an in-depth discussion of the different memory types, we refer the interested reader to [Veendrick, 2017].

Static random-access memory (SRAM) is a volatile and very fast memory. As its speed can match the processor speed, SRAM is used to build caches and processor-local scratchpad memories. Due to high area demand per bit stored, it is usually not used as the main memory except for simple, low-power microcontrollers.

Dynamic random-access memory (DRAM) is a volatile and fast memory that stores information as charge in a capacitor. Its relatively high speed and—compared to SRAM—lower area demand per bit made DRAM the predominant type of main memory in contemporary systems. In general, a DRAM access consists of two parts. First, a bigger chunk of consecutive bits, called row, is loaded into a fast SRAM-based buffer, called row buffer. Second, the requested data is taken from the row buffer. If the next access targets the currently loaded row, the first step can be omitted which reduces the latency of the access. This timing scheme motivates so-called *burst* accesses that transfer multiple consecutive words per access—roughly at the latency of a single word access. Before data from a different row can be loaded, the currently loaded row has to be written back to the DRAM cells which can increase the access latency. Furthermore, due to the leakage of the employed capacitors, their charge has to be refreshed in regular periods to

prevent data loss. An ongoing refresh can then block an incoming memory access. The overall complexity of the DRAM chips necessitates a dedicated memory controller that manages the refreshes and the required steps to perform an access.

Flash memory is non-volatile and thus used to permanently store data, e.g. the firmware code of a system. A flash chip usually stores more bits than a DRAM chip of similar size but at the cost of slower accesses. There are two main categories of flash memory architecture: NOR and NAND flash. NOR flash supports random access reads of memory words and thus can be used directly as instruction memory. NAND flash can only be accessed at a larger granularity but enables chips with high storage capacity at low cost. To reduce the average access latency, flash memory is usually combined with a buffer that can serve sequential accesses faster (burst access). All flash memories are organised in so-called blocks. A write access generally requires to first erase the surrounding block as a whole. The lifetime of the flash memory is given by the maximal number of erasures per block.

In our low-level timing analysis tool LLVMTA, we use a parametric main memory model that can be instantiated to mimic the timing behaviour of most actual main memory types. The first parameter defines the latency to access a single word, i.e. the time after the access has started until either the loaded data arrives or the data has been stored to memory. We refer to the first parameter as the *word latency* of the memory. The second parameter specifies the additional latency for any sequentially loaded/stored word to capture the behaviour of burst accesses. To account for the variable timing of accesses, e.g. due to DRAM refreshes, the third parameter provides an upper bound on this variable latency part. Throughout this dissertation, the second (third) parameter is assumed to be one (zero) if not stated otherwise. We use the term *memory latency* to refer to the overall combined latency of an access, i.e. the latency to load a whole cache line from memory into the cache.

A.2.2 Interconnect

In order to access the memory or other peripherals, the processor core needs to be *interconnected* with the memory subsystem and the peripherals. One type of interconnect is a global bus that offers a shared path to directly communicate between two components. A survey of different interconnect

structures and a common taxonomy can be found in [Anderson and Jensen, 1975] or in Appendix F of [Hennessy and Patterson, 2012].

In Figures A.1 and A.5, we consider a *bus* with two masters—the instruction and the data cache—and a single slave—the main memory. Upon concurrent instruction and data cache misses, the data access is prioritised over the instruction access as for example done in [ARM®, 2015a]. For the sake of simplicity, we assume that the bus can handle at most one access at a time and that bus accesses cannot be preempted or aborted. Furthermore, we assume that the processor, the bus, and the memory operate at the same frequency. While these assumptions are reasonable for older bus designs, contemporary bus designs are more complex. As an example, they split accesses into address and data phase and support pipelined accesses. Additionally, the processor usually operates at higher speed than the bus and memory introducing timing jitter when accessing the bus. The analysis of these more complex bus designs is, however, out of the scope of this dissertation.

In a multi-core processor, each core can act as a bus master to initiate an access to the memory. To resolve the resulting bus contention, a bus arbiter is employed to grant exclusive access to the bus. The *arbitration policy* determines which core is granted exclusive access to the bus in the next cycle. Time division multiple access (TDMA) is a time-triggered arbitration policy based on a static mapping of time slots to processor cores. TDMA achieves temporal isolation between the cores but it is not *work-conserving*, i.e. an access might not be granted although there are no other requests. In fixed-priority event-driven arbitration, each processor core is assigned a unique priority. Upon multiple requests, the core with the highest priority is granted access to the bus. Round-Robin is an event-driven arbitration policy that rotates the priorities of the cores. To this end, it remembers the core c_l that has accessed the bus last. Upon request, it grants access to the core that is next behind c_l. For a comparison of different arbitration policies and their impact on performance, we refer to [Kelter et al., 2013].

In a system with multiple masters and multiple slaves, e.g. individual memory banks or multiple peripheral devices, the bus topology with its single shared path might be inadequate. To increase performance, crossbar interconnects [Murali, 2009] with multiple parallel paths have emerged. In a full crossbar, each master has even a dedicated path to each slave which increases implementation cost.

In many-core systems, an interconnect topology with direct paths between

components becomes too expensive. Instead of direct transfers, data is transferred via multiple hops in a network on chip (NoC) [Bjerregaard and Mahadevan, 2006].

A.2.3 Cache

In general, a cache is a fast but relatively small buffer to reduce the perceived access time of a large but slow data storage. In the context of processors, a cache provides fast average access to the main memory. Unlike main memory, caches are realised in SRAM technology and located close to the core. Modern processors feature a hierarchy of caches with increasing size but decreasing speed. While there are usually separate first-level caches for code and data to allow contention-free parallel access, the other cache levels are commonly unified to allow a better overall utilisation. Throughout this dissertation, we consider a single level of separate instruction and data caches.

Despite their small size, caches significantly improve the system performance due to the principle of *locality*. It is observed, that a memory access is often followed by an access to a close-by address (*spatial* locality), e.g. when fetching straight-line code or iterating an array. Furthermore, a block that has been accessed is likely to be accessed soon again (*temporal* locality), e.g. fetching instructions of a loop.

To address spatial locality, caches operate at the granularity of cache lines that span multiple consecutive words. If a word in a cache line is requested, the whole surrounding cache line is loaded into the cache. Note that the transfer of a whole line is only slightly more expensive than the transfer of a single word due to the burst mode of main memory.

To enable fast lookups, the cache is organised as an array of independent cache sets. A part of the memory address, called index, determines the mapping of the corresponding cache line to a set. If the memory block is found within its set, called a cache hit, the data is directly transferred to the core. Otherwise, in case of a cache miss, the data is first brought into the cache by performing a main memory access and evicting another cache line.

The cache line to evict upon a miss is determined by the replacement policy. Consequently, the replacement policy tries to exploit the temporal locality while being efficient to implement. There is a multitude of replacement policies, e.g. least-recently-used (LRU), first-in first-out (FIFO),

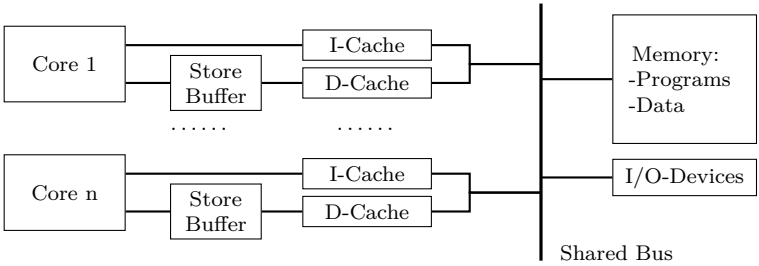


Figure A.4: Schematics of processors with store buffers, private caches, and a shared bus.

or pseudo least-recently-used (PLRU). An overview of cache replacement policies and their impact on timing predictability can be found in [Reineke, 2009].

The size of the individual cache sets is called associativity. A higher associativity offers more flexibility and a better overall cache utilisation while increasing the cost to perform cache lookups. In the extreme cases, a cache has associativity one, called direct-mapped cache, or the associativity equals the number of lines in the caches, called fully-associative.

The write policy determines how stores to memory are handled by the cache. In a write-through cache, the data is directly written to main memory. In a write-back cache, the store is performed locally in the cache and only written back to main memory once the corresponding line is evicted. If the store misses a write-back cache, the unmodified cache line is first loaded from main memory (write-allocate).

More details on caches, their taxonomy, and their impact on performance can be found in [Hennessy and Patterson, 2012].

A.2.4 Store Buffer

A store buffer is a small memory that buffers store requests generated by the processor core. Consequently, the store buffer is located between the processor core and the data cache. The store buffer allows the processor core to continue execution beyond the store instruction while the actual store is performed by the memory subsystem.

There is a variety of design choices for store buffers which influence the performance as well as the predictability of the overall system. For details on the design choices and their influence on performance, we refer to [Skadron and Clark, 1997].

In the embedded domain, store buffers with four [ARM®, 2011, 2015b] or eight [Saidi et al., 2015] entries are common. The data width of a store buffer entry ranges from word size up to the size of a cache line.

One design choice is to allow write combining of store buffer entries, also called merging or coalescing. If write combining is enabled, a store request to an address equal to the destination address of any store buffer entry is combined with this entry to a single one. While write combining can save expensive memory operations, it has implications for the memory consistency model of the system. As an example, write combining is not allowed if memory access order must be maintained, e.g. accesses to device memory [ARM®, 2011, 2015b].

The retirement order of the store buffer determines in which order the entries are actually written to memory. In the simplest case, the store buffer is drained in *first-in, first-out* (FIFO) order. The store buffer of the Kalray MPPA-256 Bostan processor aimed for timing-critical embedded systems drains the buffer in least-recently-used order [Saidi et al., 2015].

Another design choice is the retirement policy, i.e. *when* to actually drain the store buffer. One possibility is to drain the store buffer when it is full or will become full soon [ARM®, 2011]. Additionally, a store buffer entry can also be drained if it has been inside the store buffer for too long [ARM®, 2015b]. A memory barrier instruction can even explicitly drain the store buffer. Note that the actual conditions can be more complicated, but we limit ourselves to these simpler conditions for reasons of brevity.

Last but not least, the load-hazard policy defines what to do upon a load from an address of a store buffer entry. If the store buffer supports forwarding, the data from the relevant store buffer entry is forwarded to the processor. If the buffer does not support forwarding or an entry covers only part of the requested data, the colliding store buffer entries need to be drained before such a load.

A.3 Out-of-Order Pipeline

An in-order pipeline executes instructions in the order in which they appear in the program. As a consequence, when a pipeline stage stalls due to a hazard, e.g. an unsatisfied data dependency or a cache miss, the preceding stages—filled with the subsequent instructions—have to stall as well after very few cycles. The subsequent instructions thus wait for the hazard to be resolved although they might not even depend on the stalled instruction.

To gain performance, an out-of-order pipeline relaxes the condition to execute instructions in the program order. It can reorder the execution of independent instructions. As an example, it can already execute instructions while a preceding instruction waits for its dependencies to be satisfied. In addition, out-of-order pipelines are usually superscalar, i.e. they can execute instructions in parallel.

Modern pipelines commonly use Tomasulo's algorithm [Tomasulo, 1967] to implement out-of-order execution. Figure A.5 shows a brief schematic view of such a pipeline.

First, instructions are fetched from the main memory via an instruction cache and placed in an instruction queue. Next, the pipeline decodes an instruction from the queue and determines in which functional unit the instruction should execute. The pipeline issues the instruction, together with the operands that are already available, to the reservation station of the chosen functional unit. At the same time, it allocates a slot in the reorder buffer for this instruction. The *issue width* of the pipeline determines how many instructions can be issued within a single clock cycle.

The reservation stations snoop on the common data bus for operands that have just been computed. Once all operands of an instruction are available, the instruction becomes ready for execution. If a functional unit becomes idle, the corresponding reservation station chooses the instruction to execute next among the ready instructions. After the execution within the functional unit, the calculated result is put on the common data bus.

The result of an instruction, taken from the common data bus, is stored in the slot of the reorder buffer that has been allocated during the instruction issue. The reorder buffer finally commits the instructions in *program order* by writing their results to the register file. Despite being executed out-of-order, the reorder buffer maintains the program order of the instructions that are currently in the pipeline. The reorder buffer is thus essential to

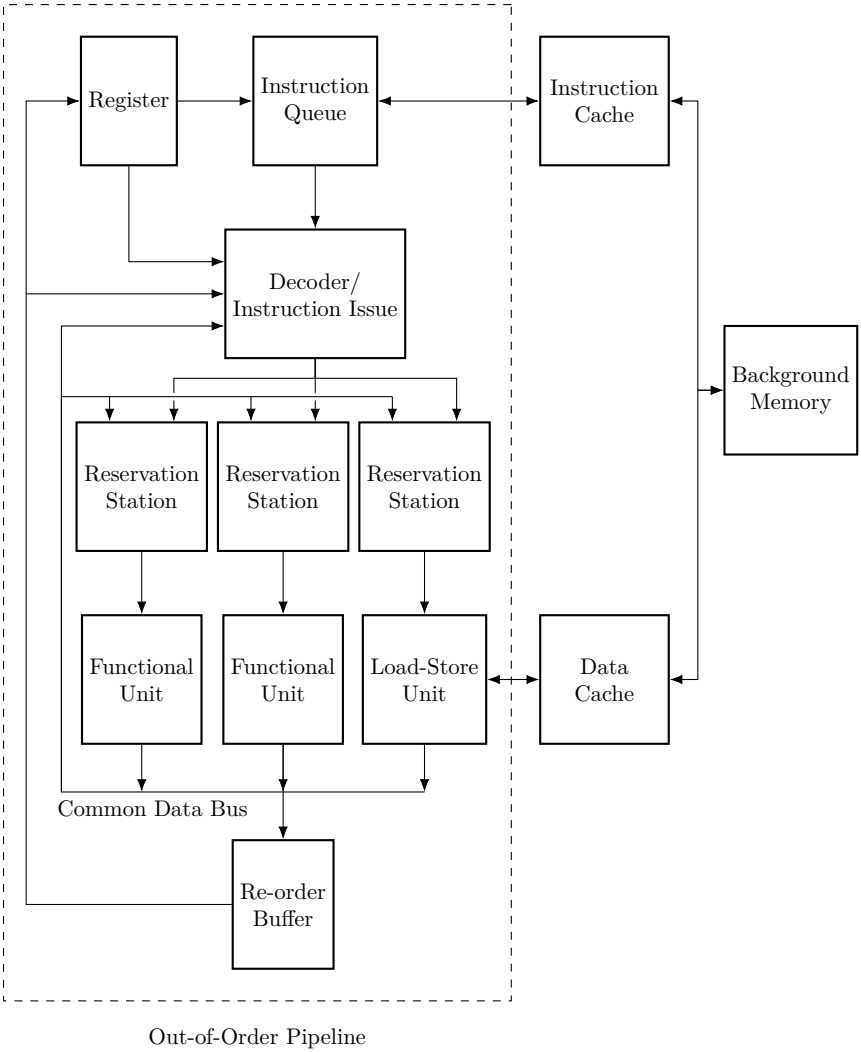


Figure A.5: Schematic view of an out-of-order pipeline according to Tomasulo's algorithm.

handle data dependencies such as write-after-write and to implement precise exceptions.

To be effective and keep the pipeline filled, the out-of-order pipeline needs to be able to continue execution beyond (conditional) branches. Besides an effective branch prediction that allows to speculatively fetch instructions, an out-of-order pipeline can speculatively *execute* instructions—in some cases even speculatively access data memory. The reorder buffer is essential for the functional correctness of speculation. Speculatively computed results are committed to the register file only if the corresponding prediction turns out to be correct. Upon a misspeculation, all speculatively executed instructions and their results are cleared and the instruction fetch is redirected to the right branch target.

For more details about out-of-order execution or hardware-based speculation, we refer to [Hennessy and Patterson, 2012].

In our low-level timing analyser LLVMTA, we model a simple out-of-order pipeline with a reorder buffer of size 8 and a 4-entry instruction queue. Multiple instructions are issued per cycle to either the Load-/Store-Unit or one of two arithmetic functional units executing instructions with variable latency. Similar to the in-order pipeline, our out-of-order pipeline employs a static branch prediction, where conditional branches that head backwards (forwards) are predicted taken (not taken). Additionally, the pipeline features speculative execution. Data memory accesses are performed non-speculatively in program order.

Appendix B

Benchmark Programs

In this section, we provide details on the benchmark programs that we have used for the evaluation of our contributions. We have three main sources of benchmarks. First, we use the Mälardalen WCET benchmark suite [Gustafsson et al., 2010] that has served as the standard benchmark in the timing analysis area for a long time. Second, we use the more recent TACLeBench suite [Falk et al., 2016] that includes most programs of the Mälardalen suite in a (heavily) edited version. In addition, TACLeBench includes new and more complex programs that are harder to analyse. We use the programs provided in the folders `app`, `kernel`, and `sequential`. Third, we use programs generated from models designed in the SCADE Suite®. The programs are generated from own models (`es_lift`, `roboDog`) or the example models delivered with the SCADE Suite® (`cruise_control`, `flight_control`, `pilot`, `digital_stopwatch`). The `trolleybus` benchmark is provided by Benjamin Meyer who developed the model in the scope of his Master's thesis at DIaLOGIKa Gesellschaft für angewandte Informatik mbH. As our timing analyser LLVMTA does currently not support programs with recursion or irreducible control-flow, we exclude those programs that make use of these features. The resulting set of C programs is listed in Table B.1.

Software for safety-critical embedded systems is often compiled *without* optimisations to ease the subsequent verification of the produced binary w.r.t. the underlying high-level model. Recently, with the emergence of formally verified compilers [Leroy, 2009], optimising compilation seems to be within reach for safety-critical systems [França et al., 2011]. To account for this development, we perform our experiments with the benchmarks compiled using CLANG 5.0.0 with optimisations (`-O2` but disabled if-conversion) and without optimisations (`-O0`), respectively.

For each program binary, we provide the number of instructions in the binary and, additionally, we determine the share of load and store instructions. The results for the non-optimised binaries are shown in Table B.1 and for the optimised binaries in Table B.2. On average, the optimised binary of a given program has 46% fewer instructions than the non-optimised binary. The share of load (store) instructions on the overall instructions decreases significantly from 28.1% to 18.0% (21.4% to 10.3%). This change in the instruction mix impacts the evaluation results as the memory hierarchy of the system is stressed more by the non-optimised binaries. Furthermore, the non-optimised binary of a given program takes—on average—2.89 times the number of processor cycles than the optimised binary, when executed on a conventional in-order pipeline as described in Appendix A.1.

Benchmark	Suite	#Instr.	#Load	%	#Store	%
`lift`	`tb/a`	950	304	32.0	166	17.5
`powerwindow`	`tb/a`	2238	658	29.4	508	22.7
`binarysearch`	`tb/k`	158	44	27.8	35	22.2
`bsort`	`tb/k`	163	39	23.9	34	20.9
`complex_updates`	`tb/k`	229	75	32.8	61	26.6
`countnegative`	`tb/k`	253	85	33.6	64	25.3
`fft`	`tb/k`	465	128	27.5	106	22.8
`filterbank`	`tb/k`	340	71	20.9	61	17.9
`fir2dim`	`tb/k`	536	179	33.4	134	25.0
`iir`	`tb/k`	204	68	33.3	50	24.5
`insertsort`	`tb/k`	187	70	37.4	36	19.3
`jfdctint`	`tb/k`	313	57	18.2	46	14.7
`lms`	`tb/k`	362	99	27.3	70	19.3
`ludcmp`	`tb/k`	516	158	30.6	89	17.2
`matrix1`	`tb/k`	197	56	28.4	51	25.9
`md5`	`tb/k`	1422	427	30.0	218	15.3
`minver`	`tb/k`	731	214	29.3	153	20.9
`pm`	`tb/k`	1841	624	33.9	442	24.0
`prime`	`tb/k`	175	37	21.1	33	18.9
`sha`	`tb/k`	1142	337	29.5	336	29.4
`st`	`tb/k`	439	136	31.0	94	21.4

adpcm_dec	tb/s	1133	378	33.4	290	25.6
adpcm_enc	tb/s	1160	369	31.8	272	23.4
audiobeam	tb/s	2418	677	28.0	491	20.3
cjpeg_transupp	tb/s	1951	577	29.6	537	27.5
cjpeg_wrbmp	tb/s	359	92	25.6	82	22.8
dijkstra	tb/s	361	107	29.6	176	48.8
epic	tb/s	2144	783	36.5	599	27.9
g723_enc	tb/s	1531	387	25.3	304	19.9
gsm_dec	tb/s	2615	653	25.0	592	22.6
gsm_encode	tb/s	5571	1465	26.3	1086	19.5
h264_dec	tb/s	1108	180	16.2	106	9.6
huff_dec	tb/s	617	165	26.7	128	20.7
mpeg2	tb/s	7393	2669	36.1	2084	28.2
ndes	tb/s	859	282	32.8	209	24.3
petrinet	tb/s	1435	615	42.9	212	14.8
rijndael_dec	tb/s	2057	829	40.3	283	13.8
rijndael_enc	tb/s	2174	858	39.5	306	14.1
statemate	tb/s	2482	1019	41.1	463	18.7
susan	tb/s	9091	3388	37.3	1389	15.3
cruise_control	sc	971	250	25.7	209	21.5
digital_stopwatch	sc	1200	308	25.7	266	22.2
es_lift	sc	1205	292	24.2	290	24.1
flight_control	sc	2989	791	26.5	801	26.8
pilot	sc	1060	264	24.9	231	21.8
roboDog	sc	2588	619	23.9	558	21.6
trolleybus	sc	7686	1974	25.7	2160	28.1
adpcm	m	1602	596	37.2	401	25.0
bs	m	87	28	32.2	23	26.4
bsort100	m	116	29	25.0	25	21.6
cnt	m	245	74	30.2	68	27.8
compress	m	778	253	32.5	174	22.4
cover	m	1082	228	21.1	222	20.5
crc	m	258	68	26.4	47	18.2
edn	m	723	222	30.7	168	23.2
expint	m	205	52	25.4	48	23.4
fdct	m	256	55	21.5	37	14.5

`fft1`	`m`	468	117	25.0	89	19.0
`fibcall`	`m`	56	12	21.4	15	26.8
`fir`	`m`	157	49	31.2	49	31.2
`insertsort`	`m`	59	12	20.3	19	32.2
`janne_complex`	`m`	77	19	24.7	19	24.7
`jfdctint`	`m`	272	48	17.6	39	14.3
`lcdnum`	`m`	116	11	9.5	25	21.6
`lms`	`m`	539	151	28.0	101	18.7
`ludcmp`	`m`	420	129	30.7	75	17.9
`matmult`	`m`	170	41	24.1	36	21.2
`minver`	`m`	625	190	30.4	133	21.3
`ndes`	`m`	799	247	30.9	206	25.8
`ns`	`m`	108	26	24.1	18	16.7
`nsichneu`	`m`	6761	3028	44.8	959	14.2
`prime`	`m`	126	21	16.7	25	19.8
`qsort-exam`	`m`	283	113	39.9	61	21.6
`qurt`	`m`	268	67	25.0	77	28.7
`select`	`m`	317	122	38.5	79	24.9
`sqrt`	`m`	123	29	23.6	27	22.0
`st`	`m`	395	112	28.4	89	22.5
`statemate`	`m`	2431	1002	41.2	453	18.6
`ud`	`m`	356	122	34.3	70	19.7
				28.1%		21.4%

Table B.1: The 79 benchmarks that we use in our experiments. They are generated from models developed in or provided by the SCADE Suite® (`sc`) and are taken from the TACLeBench (`tb`) suite (folders `app`, `kernel`, `sequential`) and the Mälardalen (`m`) suite. The instruction characterisations are obtained by compilation *without* optimisations.

Benchmark	Suite	#Instr.	#Load	%	#Store	%
lift	tb/a	632	213	33.7	88	13.9
powerwindow	tb/a	1507	302	20.0	292	19.4
binarysearch	tb/k	92	17	18.5	9	9.8
bsort	tb/k	83	13	15.7	6	7.2
complex_updates	tb/k	125	33	26.4	18	14.4
countnegative	tb/k	121	28	23.1	12	9.9
fft	tb/k	266	54	20.3	33	12.4
filterbank	tb/k	195	21	10.8	19	9.7
fir2dim	tb/k	250	49	19.6	21	8.4
iir	tb/k	98	25	25.5	9	9.2
insertsort	tb/k	131	45	34.4	24	18.3
jfdctint	tb/k	272	44	16.2	31	11.4
lms	tb/k	218	36	16.5	17	7.8
ludcmp	tb/k	260	47	18.1	18	6.9
matrix1	tb/k	91	16	17.6	9	9.9
md5	tb/k	956	155	16.2	66	6.9
minver	tb/k	347	44	12.7	25	7.2
pm	tb/k	996	225	22.6	112	11.2
prime	tb/k	120	24	20.0	13	10.8
sha	tb/k	542	81	14.9	69	12.7
st	tb/k	258	59	22.9	19	7.4
adpcm_dec	tb/s	632	209	33.1	102	16.1
adpcm_enc	tb/s	622	189	30.4	82	13.2
audiobeam	tb/s	1260	206	16.3	88	7.0
cjpeg_transupp	tb/s	831	135	16.2	101	12.2
cjpeg_wrbmp	tb/s	197	37	18.8	22	11.2
dijkstra	tb/s	218	54	24.8	83	38.1
epic	tb/s	960	199	20.7	138	14.4
g723_enc	tb/s	793	104	13.1	63	7.9
gsm_dec	tb/s	1389	203	14.6	178	12.8
gsm_encode	tb/s	3047	504	16.5	213	7.0
h264_dec	tb/s	426	91	21.4	40	9.4
huff_dec	tb/s	326	52	16.0	35	10.7
mpeg2	tb/s	3918	1120	28.6	627	16.0

ndes	tb/s	510	141	27.6	85	16.7
petrinet	tb/s	894	343	38.4	135	15.1
rijndael_dec	tb/s	1556	578	37.1	148	9.5
rijndael_enc	tb/s	1613	582	36.1	150	9.3
statemate	tb/s	1717	752	43.8	325	18.9
susan	tb/s	5117	1573	30.7	443	8.7
cruise_control	sc	495	79	16.0	51	10.3
digital_stopwatch	sc	616	93	15.1	93	15.1
es_lift	sc	491	62	12.6	101	20.6
flight_control	sc	1793	363	20.2	487	27.2
pilot	sc	548	81	14.8	59	10.8
roboDog	sc	1167	150	12.9	114	9.8
trolleybus	sc	3114	565	18.1	390	12.5
adpcm	m	901	340	37.7	146	16.2
bs	m	35	6	17.1	2	5.7
bsort100	m	61	11	18.0	6	9.8
cnt	m	101	16	15.8	11	10.9
compress	m	467	135	28.9	73	15.6
cover	m	253	8	3.2	5	2.0
crc	m	138	25	18.1	10	7.2
edn	m	426	83	19.5	56	13.1
expint	m	100	6	6.0	4	4.0
fdct	m	259	57	22.0	43	16.6
fft1	m	280	36	12.9	21	7.5
fibcall	m	25	1	4.0	1	4.0
fir	m	75	15	20.0	9	12.0
insertsort	m	45	6	13.3	11	24.4
janne_complex	m	29	1	3.4	1	3.4
jfdctint	m	245	39	15.9	29	11.8
lcdnum	m	85	5	5.9	2	2.4
lms	m	320	61	19.1	19	5.9
ludcmp	m	194	31	16.0	12	6.2
matmult	m	96	14	14.6	9	9.4
minver	m	287	36	12.5	22	7.7
ndes	m	451	99	22.0	78	17.3
ns	m	55	8	14.5	2	3.6

nsichneu	m	4295	1705	39.7	703	16.4
prime	m	81	11	13.6	8	9.9
qsort-exam	m	133	29	21.8	20	15.0
qurt	m	163	27	16.6	36	22.1
select	m	119	27	22.7	16	13.4
sqrt	m	60	7	11.7	3	5.0
st	m	239	37	15.5	19	7.9
statemate	m	1655	734	44.4	321	19.4
ud	m	155	29	18.7	12	7.7
				18.0%		10.3%

Table B.2: The 79 benchmarks that we use in our experiments. They are generated from models developed in or provided by the SCADE Suite® (`sc`) and are taken from the TACLeBench (`tb`) suite (folders `app`, `kernel`, `sequential`) and the Mälardalen (`m`) suite. The instruction characterisations are obtained by compilation *with* optimisations.

Appendix C

Additional Evaluation Results: Compositional Base Bound

C.1 Precision w.r.t. Interference Response Curve

Histograms Zero Interference

Here, we assess the precision of the compositional base bound in comparison with the interference response curve in the case that a program experiences no interference. We show the distribution of the ratios for all benchmarks in the form of histograms. For a specific interval of ratio values, the histogram shows the number of benchmarks that exhibit a ratio within this interval. We use the dot in $[a, \cdot)$ to refer to the lower bound of the subsequent interval. We show histograms for programs executed on different hardware platforms. The histograms on the left (right) show the distribution of programs compiled without (with) optimisations.

In-Order Pipeline

Figure C.1: Dual-core, blocked stores, and memory word latency of 5.

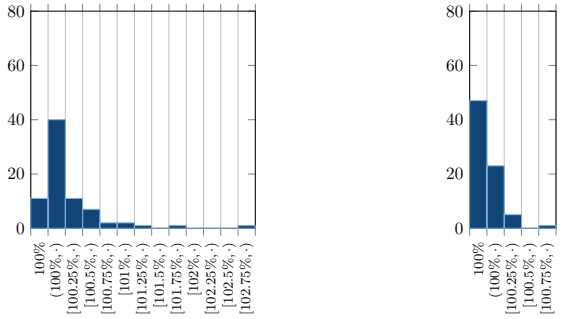


Figure C.2: Dual-core, unblocked stores, and memory word latency of 5.

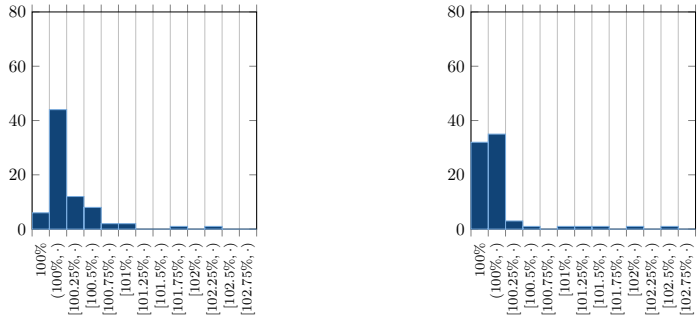


Figure C.3: Dual-core, unblocked stores, and memory word latency of 2.

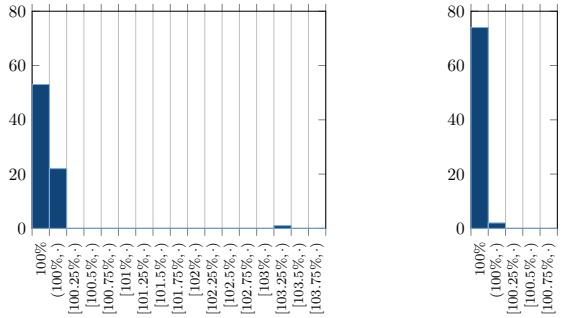


Figure C.4: Dual-core, unblocked stores, and memory word latency of 20.

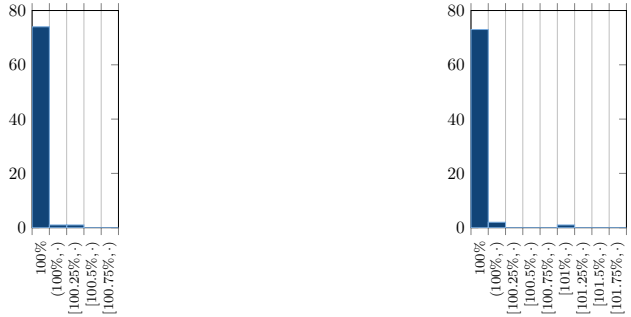


Figure C.5: Dual-core, unblocked stores, and memory word latency of 20.

Out-of-Order Pipeline

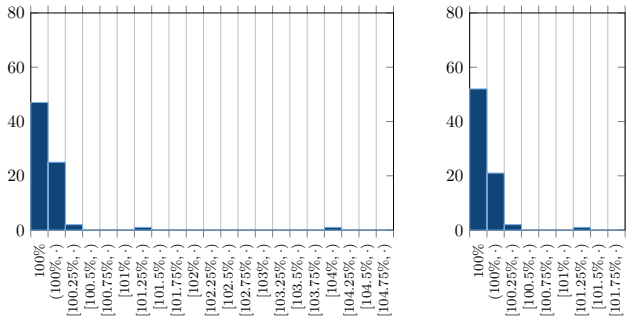


Figure C.6: Dual-core, blocked stores, and memory word latency of 5.

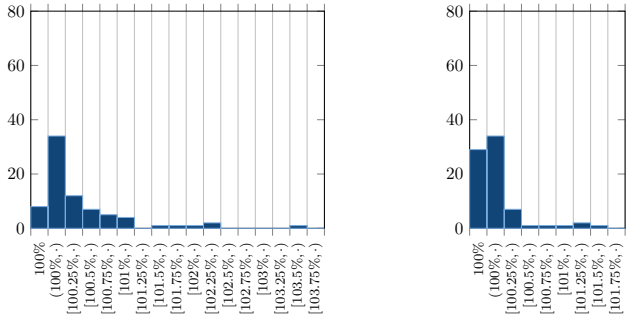
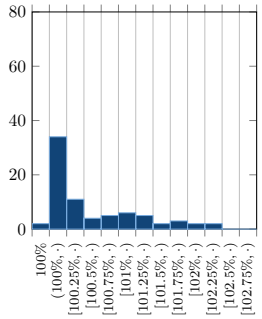


Figure C.7: Dual-core, unblocked stores, and memory word latency of 5.



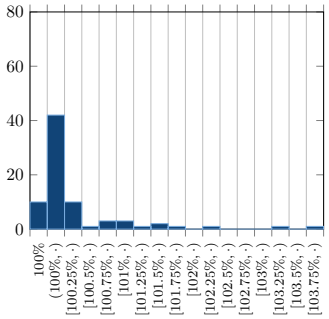
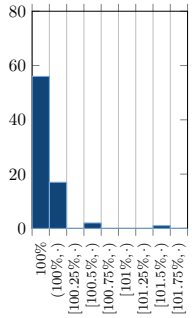


Figure C.8: Dual-core, unblocked stores, and memory word latency of 2.



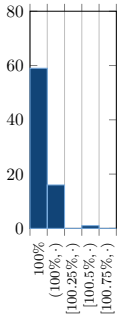
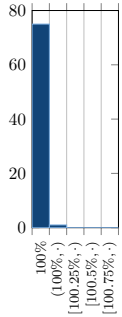


Figure C.9: Dual-core, unblocked stores, and memory word latency of 10.



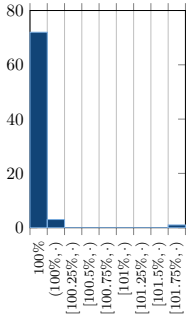


Figure C.10: Dual-core, unblocked stores, and memory word latency of 20.

Histograms Maximal Interference

Here, we assess the precision of the compositional base bound (blue bar) and the naive compositional bound (red bar) in comparison with the interference response curve in the case that a program experiences maximal interference on a shared bus with round-robin arbitration. We show the distribution of the ratios for all benchmarks in the form of histograms. For a specific interval of ratio values, the histogram shows the number of benchmarks that exhibit a ratio within this interval. We use the dot in $[a, \cdot)$ to refer to the lower bound of the subsequent interval. We show histograms for programs executed on different hardware platforms. The histograms on the left (right) show the distribution of programs compiled without (with) optimisations.

In-Order Pipeline

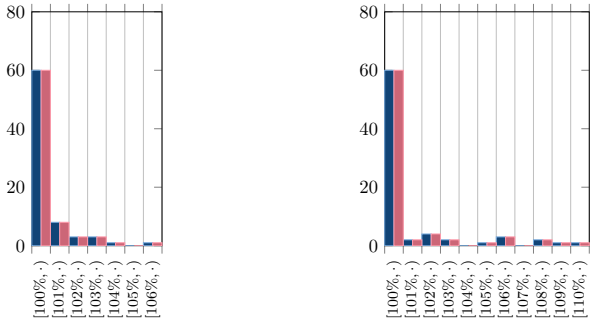


Figure C.11: Dual-core, blocked stores, and memory word latency of 5.

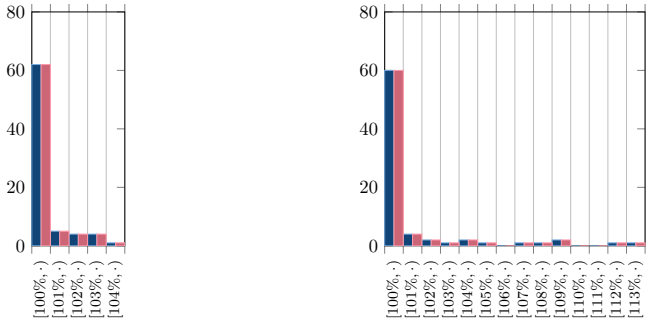


Figure C.12: Four cores, blocked stores, and memory word latency of 5.

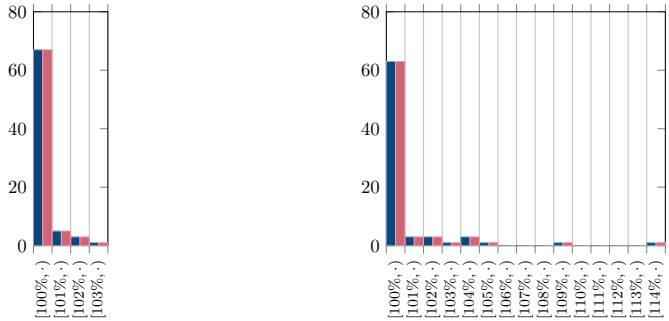


Figure C.13: Eight cores, blocked stores, and memory word latency of 5.

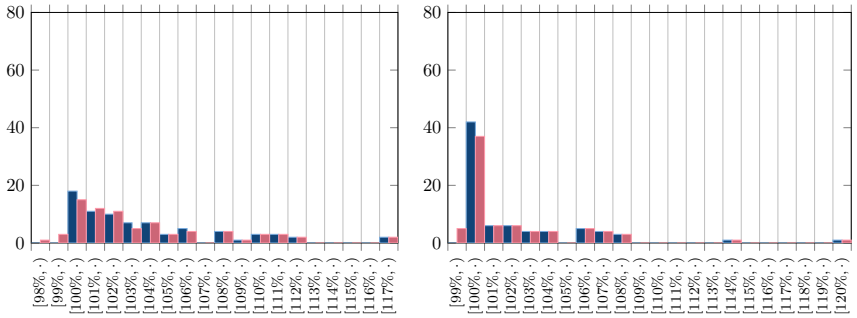


Figure C.14: Dual-core, unblocked stores, and memory word latency of 5.

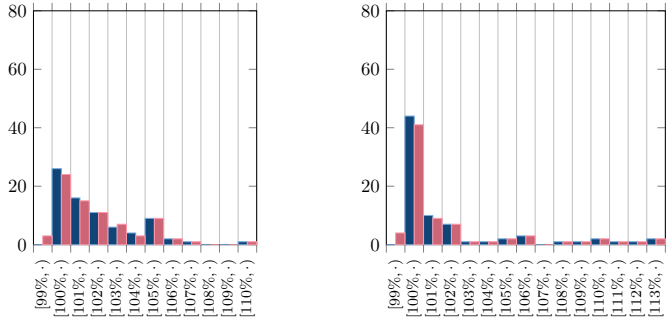


Figure C.15: Four cores, unblocked stores, and memory word latency of 5.

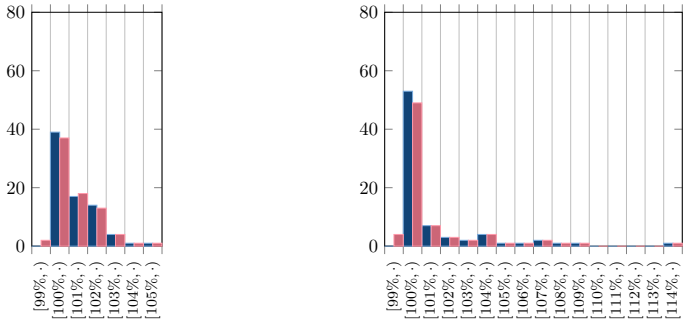


Figure C.16: Eight cores, unblocked stores, and memory word latency of 5.

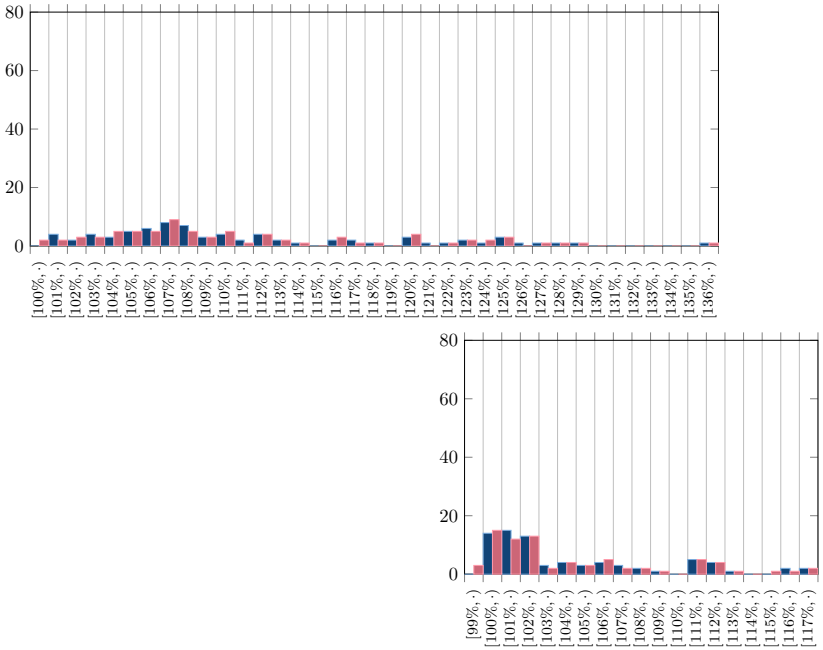


Figure C.17: Dual-core, unblocked stores, and memory word latency of 2.

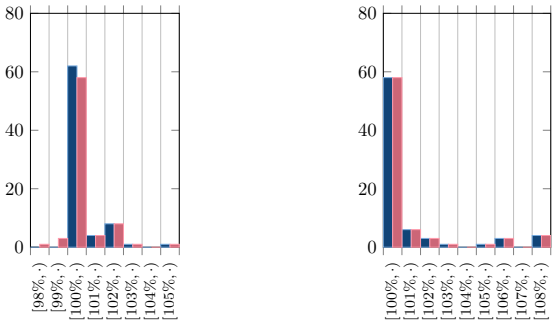


Figure C.18: Dual-core, unblocked stores, and memory word latency of 10.

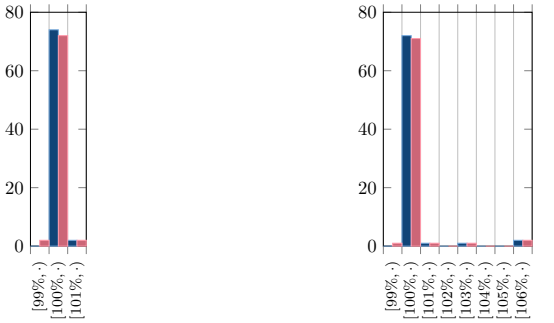


Figure C.19: Dual-core, unblocked stores, and memory word latency of 20.

Out-of-Order Pipeline

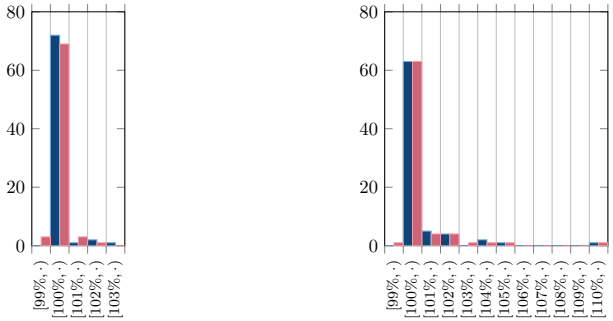


Figure C.20: Dual-core, blocked stores, and memory word latency of 5.

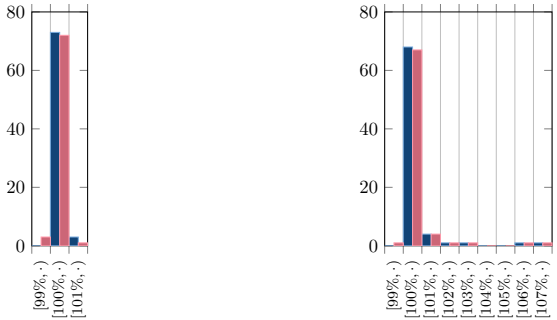


Figure C.21: Four cores, blocked stores, and memory word latency of 5.

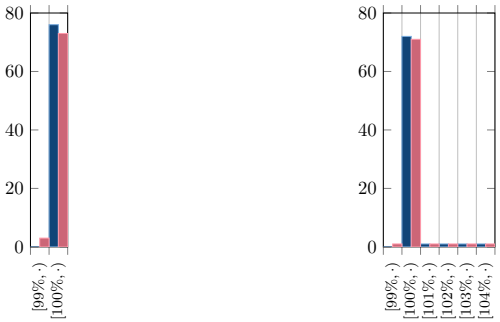


Figure C.22: Eight cores, blocked stores, and memory word latency of 5.

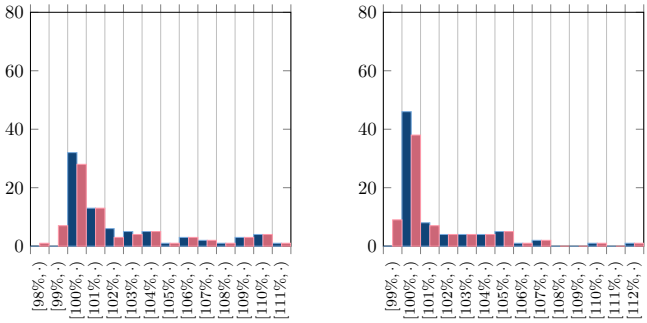


Figure C.23: Dual-core, unblocked stores, and memory word latency of 5.

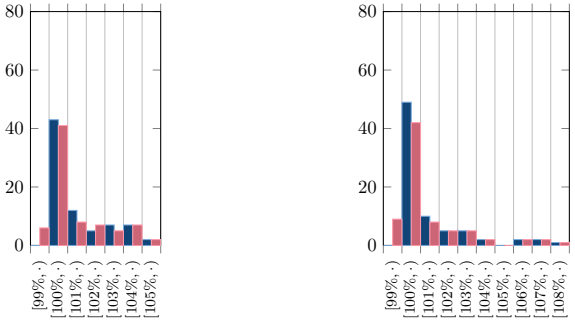


Figure C.24: Four cores, unblocked stores, and memory word latency of 5.

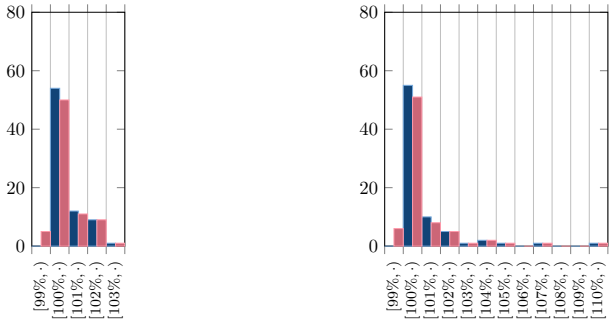


Figure C.25: Eight cores, unblocked stores, and memory word latency of 5.

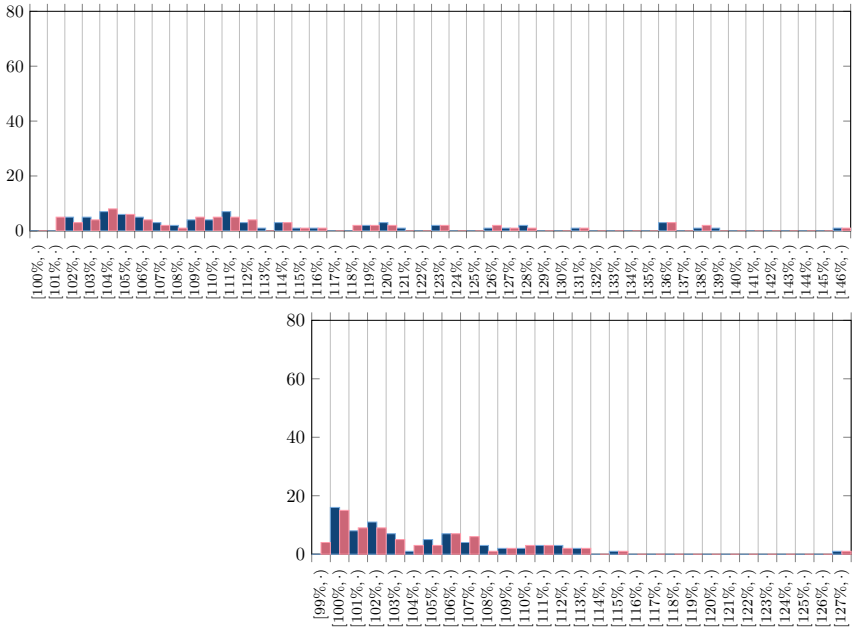


Figure C.26: Dual-core, unblocked stores, and memory word latency of 2.

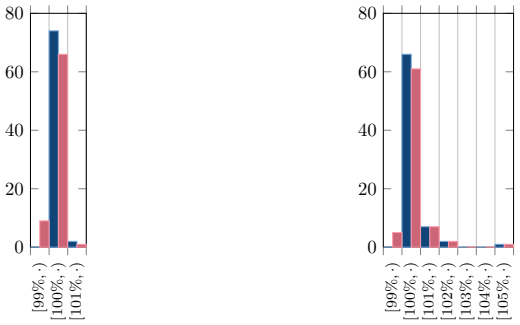


Figure C.27: Dual-core, unblocked stores, and memory word latency of 10.

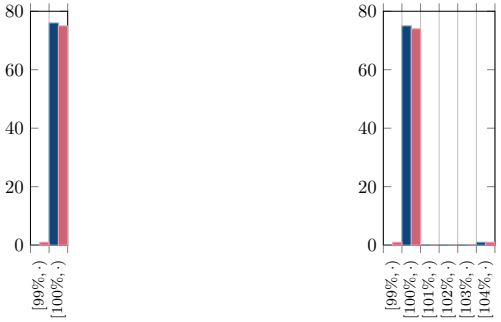
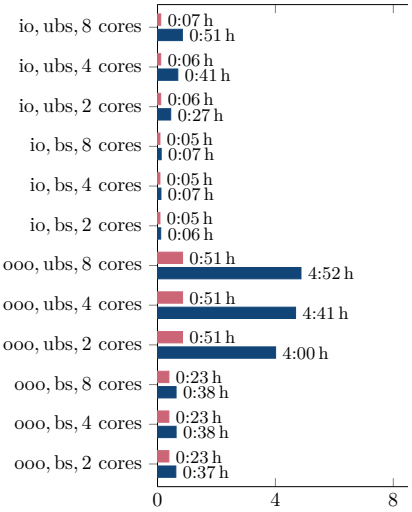
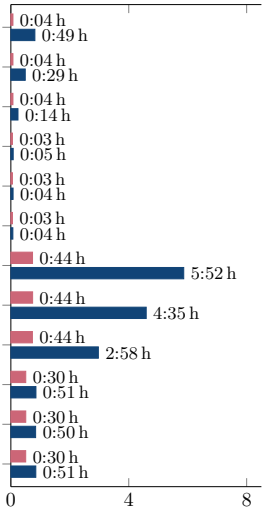


Figure C.28: Dual-core, unblocked stores, and memory word latency of 20.

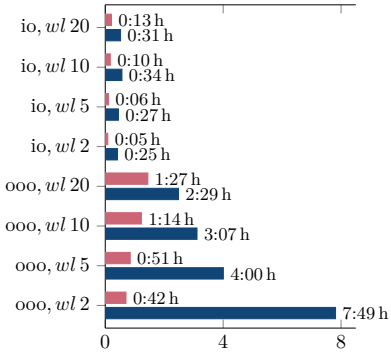
C.2 Absolute Analysis Runtimes



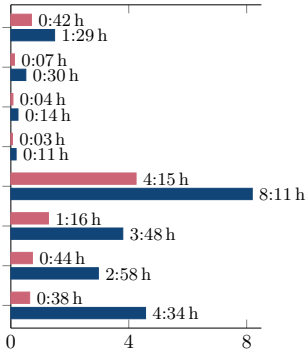
(a) Non-optimised programs. Memory word latency of 5.



(c) Optimised programs.



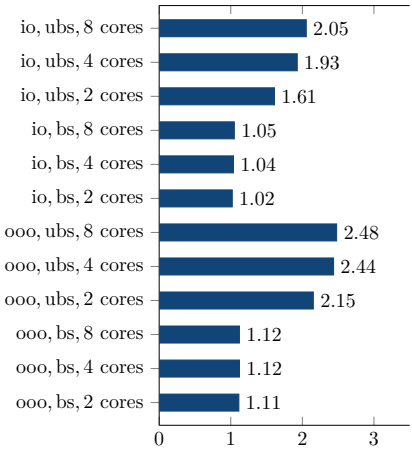
(b) Non-optimised programs. Unblocked Stores, two cores, and varying memory word latency *wl*.



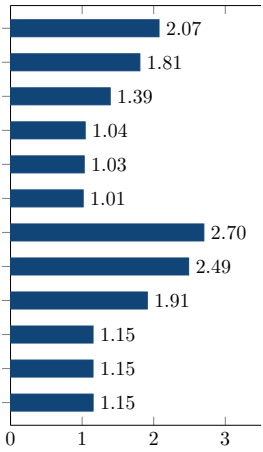
(d) Optimised programs.

Figure C.29: Runtime of compositional base bound analysis (lower, blue bar) and naive compositional analysis (upper, red bar) as sum of per-benchmark runtimes in hours. Experiments run on a single core of an Intel® Core™ i7-7700 processor, clocked at 3.6GHz.

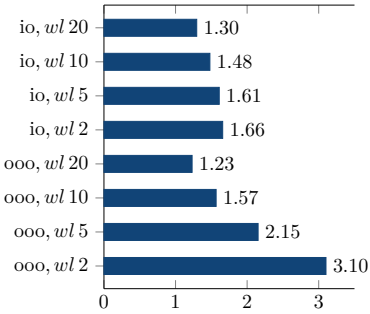
C.3 Memory Consumption



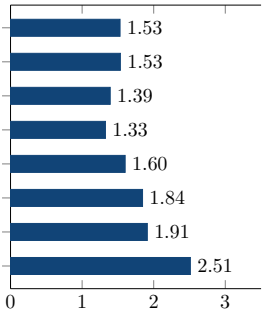
(a) Non-optimised programs. Memory word latency of 5.



(c) Optimised programs.

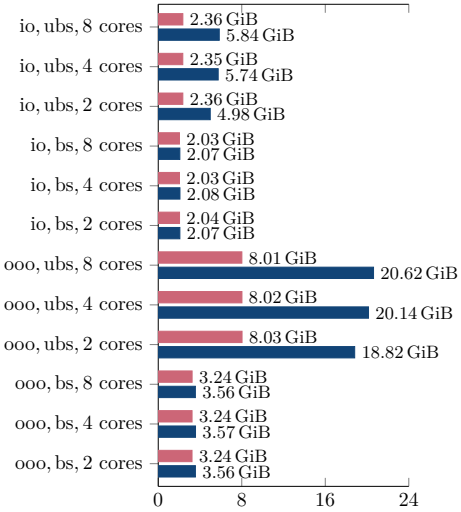


(b) Non-optimised programs. Unblocked Stores, two cores, and varying memory word latency *wl*.

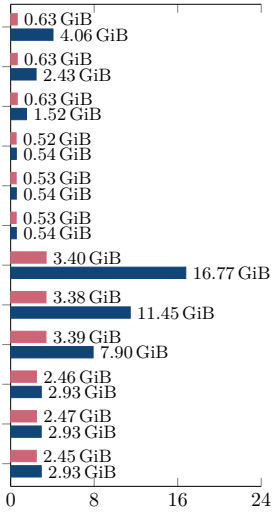


(d) Optimised programs.

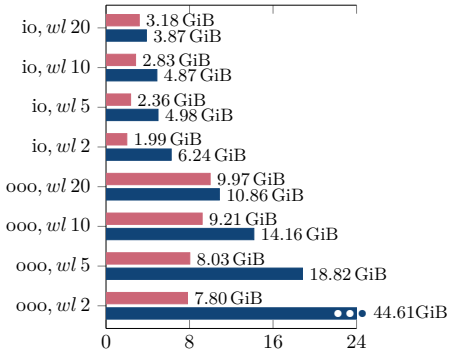
Figure C.30: Compositional base bound analysis memory consumption versus naive compositional analysis memory consumption. Geometric mean over all benchmarks.



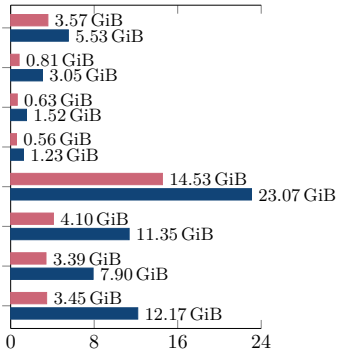
(a) Non-optimised programs. Memory word latency of 5.



(c) Optimised programs.



(b) Non-optimised programs. Unblocked Stores, two cores, and varying memory word latency *wl*.



(d) Optimised programs.

Figure C.31: Maximal memory consumption of compositional base bound analysis (lower, blue bar) and naive compositional analysis (upper, red bar) in Gibibytes.

Bibliography

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 International Conference on Concurrency Theory, CONCUR 2013, Buenos Aires, Argentina*, pages 25–43. On page 50.

Akesson, B., Molnos, A., Hansson, A., Angelo, J. A., and Goossens, K. (2010). *Composability and Predictability for Independent Application Development,Verification, and Execution*, chapter 2, pages 25–56. Springer. On page 175.

Alt, M., Ferdinand, C., Martin, F., and Wilhelm, R. (1996). Cache behavior prediction by abstract interpretation. In *Proceedings of the Third International Static Analysis Symposium, SAS 1996, Aachen, Germany*, pages 52–66. On pages 38, 61, and 162.

Altmeyer, S. (2013). *Analysis of preemptively scheduled hard real-time systems*. PhD thesis, Saarland University. On page 49.

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, RTNS 2015, Lille, France*, pages 129–138. On pages 50, 52, 54, 55, 106, and 167.

Altmeyer, S., Davis, R. I., and Maiza, C. (2011). Cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium, RTSS 2011, Vienna, Austria*, pages 261–271. On pages 53, 166, and 167.

Anderson, G. A. and Jensen, E. D. (1975). Computer interconnection structures: Taxonomy, characteristics, and examples. *ACM Computing Surveys*, 7(4):197–213. On page 190.

ARM® (2011). Cortex® - R4 and Cortex® - R4F Processor, Technical Reference Manual (revision r1p4). On page 193.

ARM® (2015a). Cortex® - M4 Processor, Technical Reference Manual (revision r0p1). On page 190.

ARM® (2015b). Cortex® - M7 Processor, Technical Reference Manual (revision r1p1). On page 193.

Åström, K. J. and Wittenmark, B. (1996). *Computer-controlled systems: Theory and design, 3rd Edition*. Prentice Hall. On page 51.

Atanassov, P. and Puschner, P. (2001). Impact of DRAM refresh on the execution time of real-time tasks. In *Proceedings of the IEEE International Workshop on Application of Reliable Computing and Communication*, pages 29–34. On pages 106 and 166.

Bachmann, O., Wang, P. S., and Zima, E. V. (1994). Chains of recurrences - a method to expedite the evaluation of closed-form functions. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation, ISSAC 1994, Oxford, UK*, pages 242–249. On page 58.

Berg, C. (2006). PLRU cache domino effects. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis, WCET 2006, Dresden, Germany*. On page 96.

Beyer, D., Henzinger, T. A., and Théoduloz, G. (2007). Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification, CAV 2007, Berlin, Germany*, pages 504–518. On pages 33 and 35.

Bjerregaard, T. and Mahadevan, S. (2006). A survey of research and practices of network-on-chip. *ACM Computing Surveys*, 38(1):1. On page 191.

Blaß, T., Hahn, S., and Reineke, J. (2017). Write-back caches in WCET analysis. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems, ECRTS 2017*. On pages 45 and 162.

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*, pages 274–279. On page 175.

Busquets-Mataix, J. V., Serrano, J. J., Ors, R., Gil, P. J., and Wellings, A. J. (1996). Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Proceedings of the Second IEEE Real-Time Technology and Applications Symposium, RTAS 1996, Boston, MA, USA*, pages 204–212. On pages 48, 166, and 167.

Calman, S. and Zhu, J. (2010). Interprocedural induction variable analysis based on interprocedural SSA form IR. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE 2010, Toronto, Ontario, Canada*, pages 37–44. On page 58.

Cassez, F., Hansen, R. R., and Olesen, M. C. (2012). What is a timing anomaly? In *Proceedings of the 12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, Pisa, Italy*, pages 1–12. On page 168.

Cazorla, F. J., Quiñones, E., Vardanega, T., Cucu, L., Triquet, B., Bernat, G., Berger, E. D., Abella, J., Wartel, F., Houston, M., Santinelli, L., Kosmidis, L., Lo, C., and Maxim, D. (2013). PROARTIS: probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems*, 12(2s):94:1–94:26. On page 159.

Chiou, D., Jain, P., Rudolph, L., and Devadas, S. (2000). Application-specific memory management for embedded systems using software-controlled caches. In *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA*, pages 416–419. On page 176.

Clarke, E. M., Grumberg, O., and Long, D. E. (1994). Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542. On page 9.

Cousot, P. and Cousot, R. (1976). Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France. On page 36.

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 Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA*, pages 238–252. On pages 9, 12, 61, and 161.

Cullmann, C. (2013). Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems*, 12(1s):40:1–40:25. On pages 45 and 162.

Cullmann, C. and Martin, F. (2007). Data-flow based detection of loop bounds. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, WCET 2007, Pisa, Italy*. On pages 44 and 161.

Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Computing Surveys*, 43(4):35:1–35:44. On pages 51 and 166.

de Dinechin, B. D., van Amstel, D., Poulhiès, M., and Lager, G. (2014). Time-critical computing on a single-chip massively parallel processor. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany*, pages 1–6. On pages 99 and 177.

Dietrich, C., Wägemann, P., Ulbrich, P., and Lohmann, D. (2017). SysWCET: Whole-system response-time analysis for fixed-priority real-time systems. In *Proceedings of the 23rd IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2017, Pittsburgh, PA, USA*, pages 37–48. On page 174.

Eisinger, J., Polian, I., Becker, B., Metzner, A., Thesing, S., and Wilhelm, R. (2006). Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *Proceedings of the 9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems, DDECS 2006, Prague, Czech Republic*, pages 15–20. On page 168.

Engblom, J. (2002). *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, Sweden. On pages 162, 168, 169, and 170.

Engblom, J. and Jonsson, B. (2002). Processor pipelines and their properties for static WCET analysis. In *Proceedings of the Second International Conference on Embedded Software, EMSOFT 2002, Grenoble, France*, pages 334–348. On page 168.

Ermedahl, A., Sandberg, C., Gustafsson, J., Bygde, S., and Lisper, B. (2007). Loop bound analysis based on a combination of program slicing,

abstract interpretation, and invariant analysis. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time Analysis, WCET 2007, Pisa, Italy*. On pages 44 and 161.

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 *Proceedings of the 16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, Toulouse, France*, pages 2:1–2:10. On pages 88 and 197.

Faymonville, C. (2015). Evaluating compositional timing analyses. Master's thesis, Saarland University. On page 173.

Flexeder, A., Mihaila, B., Petter, M., and Seidl, H. (2010). Interprocedural control flow reconstruction. In *Proceedings of the 8th Asian Symposium on Programming Languages and Systems, APLAS 2010, Shanghai, China*, pages 188–203. On page 160.

França, R. B., Favre-Felix, D., Leroy, X., Pantel, M., and Souyris, J. (2011). Towards formally verified optimizing compilation in flight control software. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems, DATE Workshop PPES 2011, Grenoble, France*, pages 59–68. On pages 91 and 197.

Gebhard, G. (2010). Timing anomalies reloaded. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, Brussels, Belgium*, pages 1–10. On page 168.

Goossens, K., Azevedo, A., Chandrasekar, K., Gomony, M. D., Goossens, S., Koedam, M., Li, Y., Mirzoyan, D., Molnos, A. M., Nejad, A. B., Nelson, A., and Sinha, S. (2013). Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow. *SIGBED Review*, 10(3):23–34. On page 175.

Graham, R. L. (1969). Bounds on multiprocessing timing anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429. On pages 101 and 167.

Granger, P. (1991). Static analysis of linear congruence equalities among variables of a program. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT 1991,*

Brighton, UK, Volume 1: Colloquium on Trees in Algebra and Programming (CAAP), pages 169–192. On page 161.

Grund, D. (2012). *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University. On page 162.

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*, pages 245–254. On page 176.

Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. (2010). The Mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis, WCET 2010, Brussels, Belgium*, pages 136–146. On pages 88, 108, and 197.

Gustafsson, J., Ermedahl, A., Sandberg, C., and Lisper, B. (2006). Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE Real-Time Systems Symposium, RTSS 2006, Rio de Janeiro, Brazil*, pages 57–66. On page 161.

Gustafsson, J., Lisper, B., Sandberg, C., and Bermudo, N. (2003). A tool for automatic flow analysis of C-programs for WCET calculation. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems, WORDS 2003, Guadalajara, Mexico*, pages 106–112. On page 161.

Hahn, S. and Grund, D. (2012). Relational cache analysis for static timing analysis. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems, ECRTS 2012, Pisa, Italy*, pages 102–111. On page 162.

Hahn, S., Jacobs, M., and Reineke, J. (2016). Enabling compositionality for multicore timing analysis. In *Proceedings of the 24th International Conference on Real-Time Networks and Systems, RTNS 2016, Brest, France*, pages 299–308. On pages 6 and 175.

Hahn, S. and Reineke, J. (2018). Design and analysis of SIC: A provably timing-predictable pipelined processor core. In *Proceedings of the*

39th IEEE Real-Time Systems Symposium, RTSS 2018, Nashville, Tennessee, USA. On page 6.

Hahn, S., Reineke, J., and Wilhelm, R. (2015a). Toward compact abstractions for processor pipelines. In Meyer, R., Platzer, A., and Wehrheim, H., editors, *Correct System Design*, volume 9360 of *Lecture Notes in Computer Science*, pages 205–220. Springer International Publishing. On pages 6, 61, 62, 75, and 169.

Hahn, S., Reineke, J., and Wilhelm, R. (2015b). Towards compositionality in execution time analysis: definition and challenges. *SIGBED Review*, 12(1):28–36. On pages 6, 25, and 171.

Heckmann, R., Langenbach, M., Thesing, S., and Wilhelm, R. (2003). The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054. On pages 163 and 173.

Hennessy, J. L. and Patterson, D. A. (2012). *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann. On pages 28, 58, 183, 188, 190, 192, and 196.

Jacobs, M. (2014). A framework for the derivation of WCET analyses for multi-core processors. Technical report, Saarland University. On page 15.

Jacobs, M. (2018). *Design and Implementation of WCET analyses—Including a Case Study on Multi-Core Processors with Shared Buses (to appear)*. PhD thesis, Saarland University. On pages 15 and 45.

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 23nd International Conference on Real-Time Networks and Systems, RTNS 2015, Lille, France*. On pages 38, 46, 58, 106, 111, 124, 125, 132, 149, 150, and 174.

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, ECRTS 2016, Toulouse, France*, pages 141–151. On page 15.

Keller, R. M. (1976). Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384. On page 7.

Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S., and Roychoudhury, A. (2011). Bus-aware multicore WCET analysis through TDMA offset bounds. In *Proceedings of the 23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal*, pages 3–12. On page 175.

Kelter, T., Harde, T., Marwedel, P., and Falk, H. (2013). Evaluation of resource arbitration methods for multi-core real-time systems. In *Proceedings of the 13th International Workshop on Worst-Case Execution Time Analysis, WCET 2013, Paris, France*, pages 1–10. On page 190.

Kelter, T. and Marwedel, P. (2017). Parallelism analysis: Precise WCET values for complex multi-core systems. *Science of Computer Programming*, 133:175–193. On pages 150 and 174.

Kinder, J., Zuleger, F., and Veith, H. (2009). An abstract interpretation-based framework for control flow reconstruction from binaries. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2009, Savannah, GA, USA*, pages 214–228. On page 160.

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 36.

Langenbach, M., Thesing, S., and Heckmann, R. (2002). Pipeline modeling for timing analysis. In *Proceedings of the 9th International Symposium on Static Analysis, SAS 2002, Madrid, Spain*, pages 294–309. On pages 38 and 39.

Lattner, C. and Adve, V. S. (2004). LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Second IEEE / ACM International Symposium on Code Generation and Optimization, CGO 2004, San Jose, CA, USA*, pages 75–88. On page 55.

Lee, C., Hahn, J., Min, S. L., Ha, R., Hong, S., Park, C. Y., Lee, M., and Kim, C. (1996). Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *Proceedings of the 17th IEEE Real-Time Systems Symposium, RTSS 1996, Washington, DC, USA*, pages 264–274. On pages 48, 166, and 167.

Leroy, X. (2009). Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115. On page 197.

Li, X., Roychoudhury, A., and Mitra, T. (2006). Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, 34(3):195–227. On pages 61, 101, 103, and 169.

Li, Y. S. and Malik, S. (1995). Performance analysis of embedded software using implicit path enumeration. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Real-Time Systems, LCT-RTS 1995, La Jolla, California*, pages 88–98. On pages 41, 42, and 164.

Lim, S., Bae, Y. H., Jang, G. T., Rhee, B., Min, S. L., Park, C. Y., Shin, H., Park, K., and Kim, C. (1994). An accurate worst case timing analysis technique for RISC processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium, RTSS 1994, San Juan, Puerto Rico*, pages 97–108. On page 48.

Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X., and Sadayappan, P. (2008). Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *Proceedings of the 14th International Conference on High-Performance Computer Architecture, HPCA-14 2008, Salt Lake City, UT, USA*, pages 367–378. On page 176.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61. On pages 47, 51, 53, and 165.

Liu, I., Reineke, J., Broman, D., Zimmer, M., and Lee, E. A. (2012). A PRET microarchitecture implementation with repeatable timing and competitive performance. In *Proceedings of the 30th International IEEE Conference on Computer Design, ICCD 2012, Montreal, QC, Canada*, pages 87–93. On pages 86, 92, 93, and 177.

Lundqvist, T. and Stenström, P. (1999). Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium, RTSS 1999, Phoenix, AZ, USA*, pages 12–21. On pages 19, 49, 101, 108, 146, 167, 169, and 172.

Lunniss, W., Altmeyer, S., Maiza, C., and Davis, R. I. (2013). Integrating cache related pre-emption delay analysis into EDF scheduling. In *Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2013, Philadelphia, PA, USA*, pages 75–84. On pages 166 and 167.

Lv, M., Guan, N., Reineke, J., Wilhelm, R., and Yi, W. (2016). A survey on static cache analysis for real-time systems. *Leibniz Transactions on Embedded Systems LITES*, 3(1):05:1–05:48. On page 161.

Maksoud, M. A. (2015). *Processor pipelines in WCET analysis*. PhD thesis, Saarland University. On page 163.

Matthies, N. (2006). Präzise Bestimmung längster Programmpfade anhand von Zustandsgraphen unter Berücksichtigung von Schleifen-Nebenbedingungen. Diplomarbeit, Saarland University. On pages 42 and 165.

Mauborgne, L. and Rival, X. (2005). Trace partitioning in abstract interpretation based static analyzers. In *Proceedings of the 14th European Symposium on Programming, ESOP 2005, Edinburgh, UK*, pages 5–20. On pages 32 and 57.

Meindl, B. and Templ, M. (2012). Analysis of commercial and free and open source solvers for linear optimization problems. Technical report, Institut für Statistik und Wahrscheinlichkeitstheorie, Vienna University of Technology. On page 59.

Miné, A. (2006). The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100. On pages 36 and 161.

Mohan, S. and Mueller, F. (2008a). Hybrid timing analysis of modern processor pipelines via hardware/software interactions. In *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2008, St. Louis, Missouri, USA*, pages 285–294. On page 170.

Mohan, S. and Mueller, F. (2008b). Merging state and preserving timing anomalies in pipelines of high-end processors. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain*, pages 467–477. On page 170.

Muntz, R. R. and Coffman Jr., E. G. (1970). Preemptive scheduling of real-time tasks on multiprocessor systems. *Journal of the ACM*, 17(2):324–338. On page 47.

Murali, S. (2009). Designing crossbar based systems. In *Designing Reliable and Efficient Networks on Chips*, pages 15–37. Springer. On page 190.

Nielson, F., Nielson, H. R., and Hankin, C. (1999). *Principles of program analysis*. Springer. On page 12.

Plazar, S., Lokuciejewski, P., and Marwedel, P. (2009). Wcet-aware software based cache partitioning for multi-task real-time systems. In *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland*. On page 176.

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 page 164.

Raymond, P. (2014). A general approach for expressing infeasibility in implicit path enumeration technique. In *Proceedings of the 14th International Conference on Embedded Software, EMSOFT 2014, New Delhi, India*, pages 8:1–8:9. On page 44.

Reineke, J. (2009). *Caches in WCET Analysis: Predictability - Competitiveness - Sensitivity*. PhD thesis, Saarland University. On pages 162 and 192.

Reineke, J. (2014). Randomized caches considered harmful in hard real-time systems. *Leibniz Transactions on Embedded Systems LITES*, 1(1):03:1–03:13. On page 159.

Reineke, J., Liu, I., Patel, H. D., Kim, S., and Lee, E. A. (2011). PRET DRAM controller: bank privatization for predictability and temporal isolation. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, Taipei, Taiwan*, pages 99–108. On page 177.

Reineke, J. and Sen, R. (2009). Sound and efficient WCET analysis in the presence of timing anomalies. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland*. On pages 168 and 173.

Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J., and Becker, B. (2006). A definition and classification of timing anomalies. In *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis, WCET 2006, Dresden, Germany*. On pages 19, 77, 98, 108, and 168.

Rival, X. and Mauborgne, L. (2007). The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems*, 29(5). On page 32.

Saidi, S., Ernst, R., Uhrig, S., Theiling, H., and de Dinechin, B. D. (2015). The shift to multicores in real-time and safety-critical systems. In *Proceedings of the 13th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands*, pages 220–229. On page 193.

Schliecker, S. and Ernst, R. (2010). Real-time performance analysis of multiprocessor systems with shared memory. *ACM Transactions on Embedded Computing Systems*, 10(2):22:1–22:27. On pages 106, 166, and 171.

Schneider, J. (2003). *Combined schedulability and WCET analysis for real-time operating systems*. PhD thesis, Saarland University, Saarbrücken, Germany. On pages 49 and 173.

Schoeberl, M., Abbaspour, S., Akesson, B., Audsley, N. C., Capasso, R., Garside, J., Goossens, K., Goossens, S., Hansen, S., Heckmann, R., Hepp, S., Huber, B., Jordan, A., Kasapaki, E., Knoop, J., Li, Y., Prokesch, D., Puffitsch, W., Puschner, P. P., Rocha, A., Silva, C., Sparsø, J., and Tocchi, A. (2015). T-CREST: time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture - Embedded Systems Design*, 61(9):449–471. On page 177.

Schoeberl, M., Chong, D. V., Puffitsch, W., and Sparsø, J. (2014). A time-predictable memory network-on-chip. In *Proceedings of the 14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, Ulm, Germany*, pages 53–62. On page 177.

Schranzhofer, A., Pellizzoni, R., Chen, 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 2011, Chicago, Illinois, USA, pages 213–222. On pages 106 and 166.

Shah, H., Huang, K., and Knoll, A. (2014). Timing anomalies in multi-core architectures due to the interference on the shared resources. In *Proceedings of the 19th Asia and South Pacific Design Automation Conference, ASP-DAC 2014, Singapore*, pages 708–713. On page 174.

Skadron, K. and Clark, D. W. (1997). Design issues and tradeoffs for write buffers. In *Proceedings of the 3rd IEEE Symposium on High-Performance Computer Architecture, HPCA 1997, San Antonio, Texas, USA*, pages 144–155. On page 193.

Smith, J. E. and Pleszkun, A. R. (1985). Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture, ISCA 1985, Boston, MA, USA*, pages 36–44. On page 99.

Sondag, T. and Rajan, H. (2010). A more precise abstract domain for multi-level caches for tighter WCET analysis. In *Proceedings of the 31st IEEE Real-Time Systems Symposium, RTSS 2010, San Diego, California, USA*, pages 395–404. On page 162.

Stärner, J. and Asplund, L. (2004). Measuring the cache interference cost in preemptive real-time systems. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2004, Washington, DC, USA*, pages 146–154. On page 47.

Stein, I. J. (2010). *ILP-based Path Analysis on Abstract Pipeline State Graphs*. PhD thesis, Saarland University. On pages 40, 42, 43, and 165.

Tan, L. (2006). The worst case execution time tool challenge 2006: The external test. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, ISoLA 2006, Paphos, Cyprus*, pages 241–248. On page 159.

Taylor, G., Davies, P., and Farmwald, M. (1990). The TLB slice - A low-cost high-speed address translation mechanism. In *Proceedings of the 17th Annual International Symposium on Computer Architecture, ISCA 1990, Seattle, WA*, pages 355–363. On page 176.

Theiling, H. (2002). *Control Flow Graphs for Real-Time System Analysis - Reconstruction from Binary Executables and Usage in ILP-based Path Analysis*. PhD thesis, Saarland University. On pages 41, 43, 160, and 164.

Thesing, S. (2004). *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University. On pages 28, 38, 39, 162, and 163.

Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1):25–33. On pages 100 and 194.

Tomiyama, H. and Dutt, N. D. (2000). Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *Proceedings of the 8th International Workshop on Hardware/Software Codesign, CODES 2000, San Diego, California, USA*, pages 67–71. On page 48.

Ungerer, T., Cazorla, F. J., Sainrat, P., Bernat, G., Petrov, Z., Rochange, C., Quiñones, E., Gerdes, M., Paolieri, M., Wolf, J., Cassé, 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(5):66–75. On page 176.

Veendrick, H. (2017). *Nanometer CMOS ICs - From Basics to ASICs*. Springer. On page 188.

Wegener, S. (2017). Towards multicore WCET analysis. In *Proceedings of the 17th International Workshop on Worst-Case Execution Time Analysis, WCET 2017, Dubrovnik, Croatia*, pages 7:1–7:12. On page 148.

Wenzel, I., Kirner, R., Puschner, P. P., and Rieder, B. (2005). Principles of timing anomalies in superscalar processors. In *Proceedings of the NASA / DoD Conference on Evolvable Hardware, EH 2005, Washington, DC, USA*, pages 295–306. On pages 168 and 169.

Wilhelm, R., Altmeyer, S., Burguière, C., Grund, D., Herter, J., Reineke, J., Wachter, B., and Wilhelm, S. (2010). Static timing analysis for hard real-time systems. In *Proceedings of the 11th International Conference on Verification, Model Checking, and Abstract Interpretation, VMCAI 2010, Madrid, Spain*, pages 3–22. On page 112.

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D. B., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P. P., Staschulat, J., and Stenström, P. (2008). The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53. On pages 3 and 159.

Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., and Ferdinand, C. (2009). Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978. On pages 171, 172, 176, and 177.

Wilhelm, S. (2012). *Symbolic representations in WCET analysis*. PhD thesis, Saarland University. On page 163.

Zimmer, M., Broman, D., Shaver, C., and Lee, E. A. (2014). Flexpret: A processor platform for mixed-criticality systems. In *Proceedings of the 20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany*, pages 101–110. On pages 90 and 177.