# Symbolic Representations in WCET Analysis

**Zur Erlangung des Grades des**
**Doktors der Ingenieurwissenschaften**
**der Naturwissenschaftlich-Technischen Fakultäten**
**der Universität des Saarlandes**

Stephan Wilhelm

Saarbrücken, November 2011

**Impressum**

Copyright © 2012 Stephan Wilhelm
Druck und Verlag: epubli GmbH, Berlin, `www.epubli.de`
ISBN: 978-3-8442-2463-4

## Abstract

Reliable task-level execution time information is indispensable for validating the correct operation of safety-critical embedded real-time systems. Static worst-case execution time (WCET) analysis is a method that computes safe upper bounds of the execution time of single uninterrupted tasks. The method is based on abstract interpretation and involves abstract hardware models that capture the timing behavior of the processor on which the tasks run. For complex processors, task-level execution time bounds are obtained by a state space exploration which involves the abstract model and the program. Partial state space exploration is not sound. A full exploration can become too expensive. Symbolic state space exploration methods using binary decision diagrams (BDDs) are known to provide efficient means for covering large state spaces. This work presents a symbolic method for the efficient state space exploration of abstract pipeline models in the context of static WCET analysis. The method has been implemented for the Infineon TriCore 1 which is a real-life processor of medium complexity. Experimental results on a set of benchmarks and an automotive industry application demonstrate that the approach improves the scalability of static WCET analysis while maintaining soundness.

## Zusammenfassung

Zuverlässige Informationen über die Ausführungszeiten von Programmen sind unerlässlich, um das korrekte Verhalten von sicherheitskritischen eingebetteten Echtzeitsystemen zu garantieren. Die statische Analyse der längsten Ausführungszeit, der sogenannten WCET, ist eine Methode zur Berechnung sicherer oberer Schranken der Ausführungszeiten einzelner, nicht unterbrochener Programmtasks. Sie beruht auf der Methode der Abstrakten Interpretation und verwendet abstrakte Modelle, die das Zeitverhalten des Prozessors erfassen, auf dem die Programme ausgeführt werden. Die Berechnung der Ausführungszeitschranken komplexer Prozessoren basiert auf der Exploration eines Zustandsraums, der sowohl das abstrakte Modell, als auch das Programm umfasst. Eine nur teilweise Abdeckung dieses Zustandsraums liefert dabei keine verlässlichen Ergebnisse. Eine vollständige Exploration ist hingegen sehr aufwändig. Symbolische Methoden, die binäre Entscheidungsdiagramme (BDDs) verwenden, sind dafür bekannt, dass sie die effiziente Abdeckung großer Zustandsräume erlauben. Die vorliegende Arbeit stellt eine symbolische Methode zur effizienten Exploration von Zustandsräumen abstrakter Pipelinemodelle im Rahmen der statischen WCET-Analyse vor. Die Methode wurde für einen realen Prozessor mittlerer Komplexität, den Infineon TriCore 1, implementiert. Ergebnisse von Experimenten mit Benchmarks sowie mit einer Anwendung aus dem Automobilbereich zeigen, dass der Ansatz die Skalierbarkeit statischer WCET-Analyse verbessert, wobei die Zuverlässigkeit der berechneten Schranken gewahrt bleibt.

## Extended Abstract

Reliable task-level execution time information is indispensable for validating the correct operation of safety-critical (or hard) embedded real-time systems. Hard real-time systems use static scheduling strategies that fail if a task misses its assigned deadline. Such failures may lead to wholesome system crashes and are therefore unacceptable in a safety-critical context. However, this type of failure can be prevented if the scheduling takes the worst-case execution times (WCETs) of all tasks into account. In practice the WCET of a task cannot be determined exactly for realistic systems. However, upper bounds of the WCET are sufficient for computing safe schedules. Actually determining such bounds is difficult because the execution time of a task depends not only on the executed program code, but also on the potential input values, and on the clock rate of the processor on which the task runs. Modern processors implement features to reduce the average execution time, e.g., pipelines and caches. Execution times on such processors also depend on the execution history and on the start state of the hardware. As a consequence, tools for WCET prediction have to cover all feasible program paths, inputs, and hardware states.

Measurement-based approaches for WCET prediction cannot guarantee full coverage and are prone to miss worst-case situations. In particular, hardware-related timing accidents like pipeline stalls and cache misses can be hard to stimulate but can cause tremendous variations in execution time. In contrast, static WCET analysis can guarantee safe upper bounds of the WCET. The state space of the hardware is covered using abstract cache and pipeline models. The employed pipeline models are finite state machines (FSMs) whose transitions correspond to processor clock cycles. Upper bounds of the execution time of a program can be obtained by counting cycles of the pipeline model. To predict timing accidents, an overapproximation of the set of reachable pipeline and cache states is computed for each program point.

Static WCET analysis only becomes computationally feasible in practice by using abstraction, which is applied to both the modeling of processor and program behavior. However, abstraction loses information which leads to uncertainty, e.g., it may not be possible to statically determine the exact address of a memory access. Furthermore, program inputs are not precisely known in advance. At the level of the pipeline model, this lack of information is accounted for by non-deterministic choices. To be safe, the analysis has to exhaustively explore all reachable states of the FSM. This can lead to state explosion making an explicit enumeration of states infeasible due to memory and computation time constraints. The problem of efficiently covering the state space of large FSMs is well-known in hardware model checking. Only the use of symbolic methods, using Binary Decision Diagrams (BDDs), allowed model checking to be successfully applied for the verification of large, complex hardware models. To this end, symbolic model checking uses an implicit encoding of the transition system and of analysis information, like sets of states in state traversal.

This work discusses how the implicit state traversal methods of symbolic model checking can be used for static WCET analysis to allow for a more efficient handling of abstract pipeline models. Static WCET analysis only scales to realistic programs

and hardware models because it carefully separates the problem of WCET prediction into several easier sub-problems. The symbolic implementation of pipeline analysis must therefore exchange results with other analyses that use different abstract representations. We show how this exchange of information can be implemented in an efficient way. Symbolic pipeline analysis also faces the specific problem that it considers the combined state space of a hardware *and* software model. This sets it apart from hardware and software model checking which only address hardware *or* software, respectively. However, symbolic methods are very sensitive with regard to the size of the model. A combined hard- and software model is huge and a symbolic analysis of such a model may therefore be inefficient. This work presents a method for decomposing the software under analysis without compromising the soundness of the approach. The decomposition allows to analyze parts of the software individually using smaller, more efficient models. Analysis information, i.e., sets of pipeline states, is translated between program parts using symbolic computations. Finally, interfacing a non-symbolic cache analysis is particularly difficult because of the interdependence between pipeline and cache. We show a way for solving this interface problem efficiently.
The described symbolic pipeline domain for static WCET analysis has been implemented in a commercial WCET tool. We present the general structure of the implementation and describe the model of the Infineon TriCore 1, an embedded processor of medium complexity that is commonly used for safety-critical applications in the automotive industry. The implementation has been evaluated using benchmark programs and an industrial automotive software for engine control. The results demonstrate that the symbolic implementation of pipeline domains improves the scalability of static WCET analysis.

# Erweiterte Zusammenfassung

Zuverlässige Informationen über die Ausführungszeiten von Programmen sind unerlässlich, um das korrekte Verhalten von sicherheitskritischen (harten) eingebetteten Echtzeitsystemen zu garantieren. Harte Echtzeitsysteme verwenden statische Verfahren zur Ablaufplanung. Diese Verfahren versagen jedoch, wenn eine Task die ihr zugewiesene Zeitschranke überschreitet. Solche Fehler können zu kompletten Systemabstürzen führen und sind daher in einem sicherheitskritischen Kontext inakzeptabel. Es ist jedoch möglich, diese Art von Fehler zu vermeiden, indem man bei der Ablaufplanung die längsten Ausführungszeiten (WCETs) aller Tasks berücksichtigt. In der Praxis ist die genaue Ermittlung der WCET eines realistischen Programms nicht möglich, jedoch sind obere Schranken der Ausführungszeit hinreichend für die Berechnung sicherer Ablaufpläne. Die tatsächliche Bestimmung solcher Schranken ist allerdings schwierig, da die Ausführungszeit nicht nur vom Programm selbst, sondern auch von den möglichen Eingabewerten und der Taktfrequenz des ausführenden Prozessors abhängt. Moderne Prozessoren reduzieren die durchschnittliche Ausführungszeit mit Hilfe von Caches und Pipelines. Bei solchen Prozessoren hängt die Ausführungszeit auch von der Ausführungsgeschichte und dem Ausgangszustand der Hardware ab. Daher müssen Werkzeuge zur Vorhersage der WCET alle zulässigen Programmpfade, Eingaben und Hardwarezustände abdecken.

Messbasierte Verfahren zur Vorhersage der WCET können keine vollständige Abdeckung garantieren und berücksichtigen häufig nicht den schlimmsten Fall. Insbesondere Verzögerungen, die durch Hardwareeffekte ausgelöst werden, beispielsweise ein zeitweises Anhalten der Pipeline oder Zugriffe auf Daten, die sich nicht im Cache befinden, können in Messungen nur schwer stimuliert werden. Solche Effekte können jedoch zu erheblichen Varianzen der Ausführungszeit führen. Im Gegensatz zu messbasierten Verfahren liefert die statische WCET-Analyse sichere obere Ausführungszeitschranken. Der Zustandsraum der Hardware wird dabei mit Hilfe abstrakter Cache- und Pipelinemodelle abgedeckt. Die dazu verwendeten Pipelinemodelle sind endliche Automaten (FSMs), deren Übergänge Prozessorzyklen entsprechen. Obere Ausführungszeitschranken eines Programms erhält man durch Zählen der Zyklen im Pipelinemodell. Zur Vorhersage hardwarebedingter Verzögerungen wird eine Überapproximation der möglichen Pipeline- und Cachezustände für jeden Programmpunkt berechnet.

Statische WCET-Analysen werden erst durch Abstraktionen auf der Ebene des Prozessormodells und des Programmverhaltens praktisch berechenbar. Solche Abstraktionen sind mit einem Informationsverlust verbunden, der zu Unschärfen in der Analyse führt. So ist es beispielsweise nicht immer möglich, die genaue Adresse eines Speicherzugriffs statisch zu bestimmen. Desweiteren sind Programmeingaben im Vorhinein ebenfalls nicht genau bekannt. Auf der Ebene des Pipelinemodells werden solche Ungenauigkeiten durch nicht-deterministische Übergänge behandelt. Um ein sicheres Ergebnis zu erhalten, muss die Analyse die erreichbaren FSM-Zustände vollständig abdecken. Die erreichbare Zustandsmenge kann dabei so groß werden (Stichwort Zustandsexplosion), dass eine explizite Aufzählung aller Zustände aufgrund von Speicherplatz- und Berechnungszeitlimits unmöglich wird. Die effiziente Abdeckung von Zustandsräumen großer FSMs ist ein bekanntes Problem aus dem Bereich des Hardware Model Checking. Erst durch die Verwendung symbolischer Methoden, basierend auf binären Entscheidungsdiagrammen (BDDs), ist es gelungen, komplexe Hardwaremodelle erfolgreich durch Model Checking zu verifizieren. Symbolisches Model Checking verwendet dabei eine implizite Darstellung, sowohl des Transitionssystems, als auch von Analyseinformation wie z.B. Mengen von Zuständen.

Die vorliegende Arbeit zeigt, wie implizite Explorationsmethoden aus dem Bereich des symbolischen Model Checking in der statischen WCET-Analyse eingesetzt werden können, um eine effizientere Behandlung abstrakter Pipelinemodelle zu ermöglichen. Statische WCET-Analyse skaliert nur deshalb auf realistische Programme und Hardwaremodelle, weil sie das Problem der WCET-Vorhersage in mehrere, jeweils einfachere, Teilprobleme zerlegt. Die symbolische Implementierung der Pipelineanalyse muss daher Informationen mit anderen Analysen austauschen, die andere abstrakte Darstellungen verwenden. In der vorliegenden Arbeit wird gezeigt, wie dieser Informationsaustausch auf effiziente Weise implementiert werden kann. Die symbolische Pipelineanalyse steht auch vor dem speziellen Problem, dass sie den kombinierten Zustandsraum eines Hard- *und* Softwaremodells betrachtet. Damit unterscheidet sie sich von Hardware und Software Model Checking, welches jeweils nur die Hardware *oder* die Software untersucht. Leider reagieren symbolische Methoden sehr empfindlich auf die Modellgröße. Da ein kombiniertes Hard- und Softwaremodell sehr groß ist, kann die symbolische Analyse eines solchen Modells ineffizient sein. Die vorliegende Arbeit zeigt eine Methode zur Zerlegung der analysierten Software, wobei die Zuverlässigkeit der Analyseergebnisse gewahrt bleibt. Die Zerlegungsmethode erlaubt es, Teile der Software mit kleineren Modellen effizienter zu analysieren. Die Analyseinformation, d.h. Mengen von Pipelinezuständen, wird zwischen den Programmteilen mit Hilfe symbolischer Berechnungen übersetzt. Schließlich bleibt das Problem der Kopplung mit einer nicht-symbolischen Cacheanalyse, welches aufgrund der wechselseitigen Abhängigkeit zwischen Pipeline und Cache besonders schwer zu lösen ist. Eine effiziente Lösung dieses Kopplungsproblems wird ebenfalls vorgestellt.

Die vorgestellte symbolische Pipelineanalyse wurde im Rahmen eines kommerziellen Analysewerkzeugs zur statischen WCET-Vorhersage implementiert. Diese Arbeit zeigt auch den allgemeinen Aufbau der Implementierung und beschreibt das Modell des Infineon TriCore 1, eines eingebetten Prozessors mittlerer Komplexität, der in sicherheitskritischen Anwendungen in der Automobilindustrie weit verbreitet ist. Die Implementierung wurde auf Benchmarkprogrammen und einer Automobilanwendung zur Motorsteuerung evaluiert. Die Ergebnisse belegen, dass die symbolische Implementierung der Pipelineanalyse die Skalierbarkeit statischer WCET-Analysen verbessert.

## Acknowledgments

First of all, I would like to thank my advisor Prof. Reinhard Wilhelm for his invaluable advice and continuous support over the last couple of years. He gave me enough freedom to let me find my way and enough guidance to get this thesis finished.

My best thanks also go to the other members of my committee: Prof. Bernd Becker, Prof. Sebastian Hack, and Dr. Daniel Grund for taking the time to review this thesis and for taking part in the defense session.

Moreover, I am grateful for the support of many people at AbsInt GmbH and at the Compiler Design Lab at Saarland University. Christian Ferdinand allowed me to dedicate some of my working time at AbsInt to research. Daniel Kästner and Reinhold Heckmann both did an excellent job of patiently proof reading several versions of the manuscript. Björn Wachter and Christoph Cullmann had a part in getting the results of this work published at international workshops and conferences. Florian Martin, Michael Schmidt, and Henrik Theiling introduced me to the depths of AbsInt's static WCET analysis tool chain. Frank Fontaine, Gernot Gebhard, Markus Pister, Marc Schlickling, and Ingmar Stein shared with me their knowledge of pipeline models. Thanks also to Nicolas Fritz, Christian Hümbert, Marc Langenbach, Philipp Lucas, Stefana Nenova, Martin Sicks, and Stephan Thesing for their support and for contributing to a pleasant working atmosphere. I sincerely apologize for having forgotten to mention anyone to whom acknowledgment is due. Rest assured that I am grateful for every bit of support and encouragement that I ever got.

Finally, I would like to thank my family and friends. Their love and support helped me to keep up the motivation for this long-term project. The most special thanks go to my parents Rita and Heribert and – last but not least – to my wife Nicole.

– For Ronja, who has changed my life. –

# Contents

# CHAPTER 1

# Introduction

## 1.1. WCET Analysis for Hard Real-Time Systems

Over the last two decades, computers have found their way into many objects of daily life, including some whose origins precede the computer era. Cars are a very good example. In the seventies of the last century, an automotive engine wasn't much more than four cylinders powering a driveshaft. Today, all modern engines use a computer for precise control of the engine speed, temperature, and the injection pump, in order to reduce fuel consumption, and to fulfill emission standards. Such computers, together with the software that is running on them, are known as *embedded systems*. They are not only in cars; modern airplanes, cell phones, and portable MP3 players also rely on embedded computers – and *rely* means that they will not work properly if the embedded system fails. People may not care much if their MP3 player chokes on some particular track or if their cell phone breaks down in the middle of an unpleasant conversation. But a person who is sitting in an airplane that approaches the runway with landing speed wants to be sure that all relevant embedded systems in the plane are working correctly. Therefore, many embedded systems in airplanes and cars are considered to be *safety-critical* – lives may depend on their proper operation.

Safety-critical embedded systems obviously have to satisfy high quality requirements. For example, the flight control system of an airplane should never perform a division by zero which would halt the processor. Many embedded systems are also critical with respect to the execution time; e.g., an engine control system must perform its computations fast enough, even if the engine is running at full speed. Embedded systems with execution time constraints are termed *real-time* embedded systems. Systems with execution time constraints that are safety-critical are called *hard* real-time systems. For these systems, one must not only prove the absence of critical run-time errors – like division by zero – but also ensure that the system always reacts in-time. The question is: how can such properties be proved? In the past, engineers verified the safety of their constructions by testing; either by putting critical loads onto their construction, or by running long-term tests. For mechanical systems, this approach is still valid, although it is increasingly replaced by computer simulation for reasons of cost. However, for safety-critical embedded systems, testing is no longer sufficient because of two reasons: the state space of the embedded soft- and hardware is too large to admit an adequate test coverage, and second, it is usually not possible to choose a limited set of inputs that stimulates the critical executions. This problem is well-known in computer science; it is equivalent to the insight that all interesting questions about the concrete semantics of a program are undecidable. A solution for answering – at least some – questions about the undecidable semantics of a program exists in the form of *abstract interpretation*. A sound abstract interpretation is a *static analysis* that replaces the undecidable concrete semantics by abstract semantics that over-approximate any concrete execution. The gained computability is paid for by a loss of precision; the analysis may consider spurious executions that do not correspond to any concrete execution. However, abstract interpretation is also exhaustive, i.e., it never misses a possible execution, and therefore provides full coverage over all inputs and execution paths. These two properties – incompleteness and full coverage – are often summarized by the statement that abstract interpretation only errs on the safe side. Hence, abstract interpretation can be used to *prove* properties of software. Static analysis tools based on abstract interpretation recently found their way into the mainstream avionics and automotive industry and are now considered to provide state-of-the-art solutions for the verification of safety-critical, embedded real-time systems. As such, their use is increasingly encouraged by certification authorities, particularly in the civil avionics industry, where the certification standard DO-178B [Rad] requires the use of state-of-the-art technologies to prove the absence of errors in safety-critical systems. A good example of a static analysis tool that is used in the avionics industry is the Astrée static analyzer. Astrée can be used to prove the absence of certain run-time errors – such as division by zero, invalid pointer dereference, or array out-of-bound accesses – on synchronous command-and-control programs, written in ANSI C. Synchronous command-and-control programs consist of a main loop that controls the execution of system tasks using a static schedule. They are often found in safety-critical embedded real-time systems, such as the flight-control system of an aircraft. Obviously, the proper operation of such a system not only requires that it is free of run-time errors, but also that each task meets its deadline.

The problem of giving guarantees about the timeliness of a task is known as *worst-case execution time* prediction, or WCET prediction, for short. Because of the already mentioned lack of coverage, measurements – which are the equivalent of testing for WCET prediction – are not an option when it comes to hard, i.e., safety-critical, real-time systems. The inadequacy of measurements for the purpose of WCET prediction is visualized by Figure 1.1. Taking measurements means taking samples from the set of possible executions. In this set, the variance is usually large and the values are not evenly distributed. This is partly because of the range of inputs and

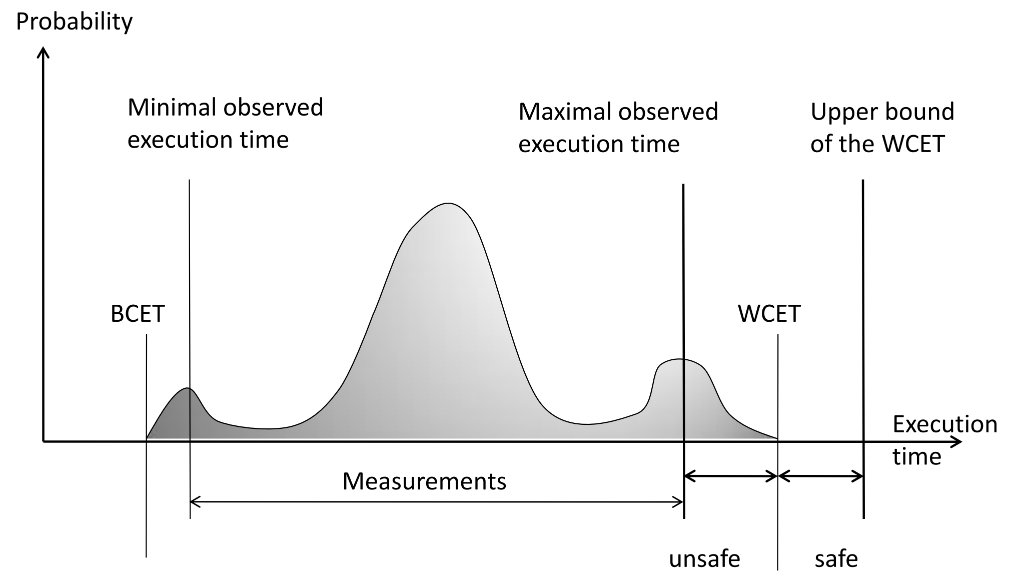


**Figure 1.1.:** Typical distribution of execution times over the execution space of a task. Measurements underestimate the WCET whereas static analyses provide safe upper bounds.

different operating modes but also because the execution time of modern hardware, featuring caches and complex pipelines, is very sensitive with respect to the start state and execution history [HLTW03, Rei08]. There is only little chance for observing unusually small or high execution times. Even worse, virtually all observed values *underestimate* the worst-case, i.e., measurements err on the unsafe side. Because the difference between the worst-case measurement and the actual worst-case is unknown, adding "safety" margins to account for underestimations, is not safe; and even if the results overestimate the worst-case, they are often not precise enough. For the analysis of single, uninterrupted tasks, static WCET analysis is a safe alternative. It provides full coverage and computes *upper bounds* on the execution time that hold for all possible executions.

## 1.2. Related Work and Contribution

Research on formal verification methods usually falls into one of the following categories: abstract interpretation, model checking, and theorem proving. Model checking and theorem proving have been successfully used for hardware verification [JM01, CGH$^+$93, CRSS94] and there have been efforts to integrate both methods

[Ber02]. In contrast, abstract interpretation originates from the area of software verification where it is successfully used, e.g., to prove the absence of run-time errors [BCC+03]. Despite recent attempts to use model checking for software verification [BHJM07, BLQ+03, WL04] there has been little exchange between the two communities.

Static WCET analysis is an area that has contact points with both, hardware and software verification. A major research effort for static WCET tools has been made by Reinhard Wilhelm's group at Saarland University. They tackled the problem from the software-oriented perspective of abstract interpretation, and developed an approach that decomposes the task into several sub-problems [FHL+01, HLTW03]. Each sub-problem is solved by a dedicated analysis. Abstract hardware models of the cache and the pipeline are used to predict low-level timing effects [Fer97, The04]. The computation of the worst-case program path is based on integer linear programming [The02]. While this approach is nowadays well-established in industry [TSH+03, WEE+08], the use of model checking for WCET analysis has been a subject of controversial discussion between researchers.

Early work on model checking for static WCET analysis focused on synchronous programs, assuming a single-cycle execution model [LS03, LSM03]. This execution model does not hold for modern processor hardware using pipelined execution and caches to improve the average execution time. More recent research takes the timing effects of the hardware into account [DOT+10], but so far addresses only the most simple pipelines, cannot handle floating-point computations, and suffers from scalability problems. At the same time, WCET tools based on abstract interpretation analyze industrial-level software, running on complex, superscalar pipelines with caches. Thus, it seems that there is a tremendous gap between model checking and abstract interpretation when it comes to WCET analysis. In 2004, Reinhard Wilhelm published a paper that discusses the challenges that model checking faces in WCET analysis [Wil04]. His major argument is still not disproved: the combined state space of program paths, variable values, pipeline, and cache state is too large to admit an exhaustive traversal without using dynamic abstraction. The reply by Metzner [Met04] misses the point when he discusses the imprecisions in abstract cache analysis [Fer97], and proposes improvements by model checking. Abstract interpretation deliberately loses precision to gain computability, and experimental results show that the computed WCET bounds are tight enough in practice [TSH+03, SLH+05].

Despite its undeniable success, static WCET analysis based on abstract interpretation sometimes runs into problems at its contact points with hardware verification. In particular, the analysis of the pipeline timing requires an exhaustive traversal of the reachable state space of the pipeline model; for pipelines with large state spaces, the analysis can become infeasible [The04]. The problem of state explosion is well-known in model checking. Symbolic representations, usually based on binary decision diagrams [Bry86], have significantly improved the situation because they admit both an implicit encoding of the transition system and of analysis information like sets of states in state traversal [McM92]. This has enabled the analysis of hardware designs

with large state spaces [BCM+90]. Although model checking alone may not be the adequate approach for WCET prediction, some of its methods can be exploited to improve static WCET analysis.
The research that is presented here focuses on improving the efficiency of pipeline analysis in cases where large sets of pipeline states have to be considered. To this end, Chapter 6 presents a novel approach for implementing pipeline analysis using a symbolic representation. The approach combines ideas from the world of model checking with the advantages of static program analysis based on abstract interpretation. Chapter 7 describes the implementation of a framework for symbolic pipeline analyses and a pipeline model of the Infineon TriCore 1 [AG08, Zar01]. Experimental results that have been obtained with this model show that the approach is indeed more efficient in practically relevant scenarios. A summary of the two chapters has been published in [WW09].[1] The integration with cache analysis has been published in [WC10]. Further contributions are found in Section 5.2 and Section 7.3. Section 5.2 gives a more profound presentation of pipeline analysis than what is found in overview publications like [FHL+01]. At the same time, the description and the given proof of soundness are more general than the presentation in [The04], which describes the pipeline model of a specific processor and proves its correctness. The state explosion problem in WCET analysis, which has first been mentioned in [The04], is discussed in detail since it represents the motivation for the use of symbolic methods. Finally, Section 7.3 discusses several practical consequences that arise from the use of symbolic methods for pipeline analysis. These insights have not been published before.

## 1.3. Overview

This work is structured into 8 chapters, including this introduction. Here is an overview of the contents of the remaining chapters:

*Chapter 2* gives an overview of WCET analysis including various system-level and task-level analysis methods. We discuss the existing research and show how task-level and system-level analyses interact.

*Chapter 3* presents the fundamentals of static program analysis. It discusses the design of abstract domains for static analyses, the actual computation of semantic invariants, and the underlying theory of abstract interpretation.

*Chapter 4* gives a similarly fundamental introduction into symbolic state traversal for sequential circuits. It comprises a discussion of binary decision diagrams as a data structure for the compact representation of Boolean functions.

*Chapter 5* presents the general design of static analysis tools for computing upper bounds on the execution time of tasks. It shows how the problem can be decomposed into several analyses and how these analyses interact. Pipeline domains based on abstract hardware models are discussed in detail.

[1]The idea first appeared in [Wil05]. Intermediate results were published in [WW07].

*Chapter 6* shows how pipeline domains can be represented symbolically using binary decision diagrams. Further, it details how such a domain can be integrated with a framework for static WCET analysis.

*Chapter 7* describes the implementation of a framework for symbolic pipeline analyses, and the implementation of a model of the Infineon TriCore 1. Several experiments that compare the performance of the symbolic implementation with a semantically equivalent explicit-state implementation are reported. Further experiments with the pipeline model of the Motorola/Freescale MPC 755 study the efficiency of the cache analysis integration. The chapter is concluded by an experience report about practical implications of the approach.

*Chapter 8* summarizes the achievements of this work. Finally, we give advice regarding the application of the approach in commercial industrial-strength WCET analyzers and propose topics for future research that could lead to further improvements of our method.

# CHAPTER 2

# Related Work

The question of giving guarantees about the worst-case execution times of software has been around for more than 20 years. Since then, the interest in this matter has continuously increased. Notably the advent of safety-critical embedded real-time systems, like fly-by-wire in civil avionics, led to intensified research efforts. This research has been conducted on two levels: the task level and the system level. Task-level WCET bounds are used as inputs for tools that determine properties on the system level, most prominently schedulability and worst-case response time (WCRT) guarantees. The following overview of related work focuses on task-level WCET analysis because that part is directly related to the subject of this thesis. We present a number of approaches and discuss their advantages as well as drawbacks. Existing uses of symbolic methods in WCET analysis are also covered. To set the scene, we begin with a brief description of the three major applications that use task-level WCET information: schedulability tests, WCRT analysis, and WCET-aware compilers.

## 2.1. Schedulability, WCRT Analysis, and Compilers

The software of embedded systems is usually composed of several tasks. Each task implements a sub-function of the system and has to be executed periodically. The period that is assigned to each task depends on the characteristics of the controlled physical system, e.g., the sample rate of a sensor. For the correct operation of the whole software the tasks may have to be executed in a certain order. The execution of each task may also be subject to further constraints, e.g., a task may only run after a certain point in time, or it must finish execution before a certain point in time. The latter constraint is called the task's *deadline*. Executing embedded software in a correct manner that respects its execution constraints and the dependencies between its tasks is a *scheduling* problem. The task schedule determines the order in which the tasks run.

The computation of task schedules is based on the notion of *priorities*. Each task is assigned a priority such that these priorities establish a total order on the set of tasks. From the set of tasks that are *ready*, i.e., that have no more unfulfilled dependencies, the scheduler selects the highest priority task to execute next. The priority of a task may either be fixed, or determined at run-time based on certain conditions. An example of a strategy for assigning fixed priorities is *rate monotonic* scheduling [LL73]. It determines the priorities of tasks by the lengths of their periods. The task with the smallest period is assigned the highest priority An example of a dynamic scheduling strategy is called *earliest deadline first* [Liu00]. It assigns the highest priority to the task with the nearest deadline. Safety-critical embedded systems usually rely on static schedules that are computed offline based on fixed scheduling strategies.

A schedule that has been computed by a given strategy, either fixed or dynamic, is called feasible if it respects all constraints and dependencies between the tasks. All scheduling strategies guarantee that a feasible schedule can be computed if certain conditions are fulfilled. The satisfaction of these conditions can be checked by a so-called *schedulability test* [BW90, Liu00]. These schedulability tests require information about the *worst-case execution time* (WCET) of all tasks. Hence, the knowledge of upper bounds on the WCET of all tasks of a real-time system is crucial to prove its correct operation.

WCET bounds for tasks are also required to obtain *worst-case response times* (WCRTs) of entire real-time systems. WCRTs can be computed by so-called *system-level* timing analysis tools, e.g., SymTA/S [HHJ$^+$05]. Beside task-level WCET bounds, these tools also take information about possible interrupts and their priorities into account. The WCRT can either be computed on the level of a single application running on one microprocessor, or on the network-level if considering embedded systems that are composed of several interacting applications running on multiple processors that are connected by a bus using protocols like CAN [ISO04] or FlexRay [Alt01].

In contrast to the class of schedulability tests and WCRT analysis tools that we discussed so far, Bernat et al. propose probabilistic methods for obtaining schedulability and response time guarantees [BCP02, BBB03]. These methods are convenient for soft real-time systems where safe WCET bounds are not required. They are not suited for safety-critical hard real-time systems because they are neither safe nor sound.

*WCET-aware compilers* are another, more recent, class of applications that rely on task-level WCET bounds [Fal09, LM09]. The idea is to use WCET information for guiding compilation strategies that reduce the worst-case instead of the average-case execution time. An example of such a tool is the `WCC` compiler infrastructure [FL10]. It implements a feedback loop with the `aiT` WCET tool [Abs00] in order to evaluate the WCET effect of already performed optimizations and to asses further optimization potential from the WCET bounds of individual basic blocks and routines.

## 2.2. Determining Task-Level WCET Bounds

It is generally not possible to predict execution times for programs since this would correspond to solving the halting problem. Real-time systems however adhere to restrictions which guarantee that these programs always terminate. That is, all loops and recursions are limited and the recursion and loop bounds are known in advance.[1] There are two classes of methods for obtaining WCET bounds for real-time tasks:

- *Dynamic* methods derive WCET estimates from observing concrete program executions. They are also referred to as *measurement-based* methods.
- *Static* methods determine WCET bounds by applying analysis methods to the program code without executing it.

All methods make pessimistic assumptions in order to approximate the worst-case behavior. *Reliable* WCET bounds – as required for designing hard real-time systems – can only be given by methods which are safe and sound. A method is *safe* if it only uses assumptions which over-approximate the worst-case, i.e., if it errs only on the safe side. A method is also *sound* if it covers all possible executions. For WCET analysis this means covering the product of all possible inputs, program paths, and hardware states.
The following two subsections give an overview of existing dynamic and static methods for computing task-level WCET bounds. The presentation generally proceeds from the simple to the more advanced methods and tries to illustrate the historic research development without covering all of the available tools. A broad and very detailed overview of the state of the art in WCET analysis methods and tools can be found in the overview article [WEE$^+$08].

### 2.2.1. Dynamic Methods

The simplest dynamic method for obtaining execution time information for a task is by measuring several executions of the whole task with varying inputs. The measurements can either be performed by adding instrumentation code to manipulate hardware timers, or by the use of a logic analyzer to observe certain signals (like fetching a certain instruction or writing to a special address). The drawback of this method is that the variance in the set of executions is usually large and the values are not evenly distributed. This is partly because of the range of inputs and different operating modes but also because the execution time of modern hardware, featuring caches and complex pipelines, is very sensitive with respect to the start state and execution history [HLTW03, Rei08]. There is only little chance for observing unusually small or high execution times. Even worse, virtually all observed values *underestimate* the worst-case, i.e., measurements err on the unsafe side. Because the difference between the worst-case measurement and the actual worst-case is unknown, adding

[1]However, experience from industrial practice shows that obtaining safe and precise recursion and loop bounds often requires a fair amount of time-consuming manual analysis.

"safety" margins to account for underestimations is not safe; and even if the results overestimate the worst-case, they are often not precise enough.
Despite the inherent lack of safety, dynamic (or measurement-based) methods can be very useful for analyzing tasks with so-called *soft*, i.e., not safety-critical, WCET constraints. The main advantage over static methods is that they do not require the construction of a hardware model, which can be expensive and time consuming depending on the complexity of the modeled hardware. However, measurement-based methods require additional tracing hardware instead. Recently, built-in debugging and tracing interfaces, e.g., the Nexus interface [O'K00], have become increasingly popular for microprocessors targeting the embedded market. The Nexus interface is very convenient to use but the employed trace buffers are too small to measure the execution of a whole task. This restriction brought forth a new generation of measurement-based timing analysis tools. These tools measure the execution times of many small snippets of the task. The individual execution times are combined to a global WCET estimate, e.g, using integer linear programming [WKRP08, Sta09], or probabilistic methods [BCP02, BBN05].
Despite all efforts to combine snippet execution times safely, these tools suffer from the same lack of soundness as full-blown measurements do; it can therefore not be guaranteed that the given WCET estimates are an upper bound of the actual WCET. At the same time, it has been reported that the combination of snippet execution times often leads to high overestimations of the WCET in practice [BBK$^+$06]. Recent efforts to take context information for the measured snippets into account [SM10] may improve on the precision issue to a limited degree.

## 2.2.2. Static Methods

A simple static approach to obtain WCET estimates on the source-code level is the use of *timing schemes* which has been proposed by Shaw [Sha89]. It works for any imperative high-level programming language and assumes that constant execution times can be assigned to atomic instructions. For straight-line code, the execution times of the atomic instructions are added up. The execution time of a conditional is given by the maximum over all cases. Loops and recursions are handled by multiplying an iteration or recursion bound with the cost of the loop or function body.
Shaw's timing schemes have several drawbacks. First, timing schemes often give very coarse WCET bounds. For example, not all iterations of a loop have the same execution time. Different iterations may execute different paths through the loop body. Hence, simply multiplying the loop bound with the worst-case iteration significantly overestimates the actual WCET in many cases. Another problem is that for tasks running on modern computer hardware it is not safe to assume that the execution time of an atomic instruction is constant. In fact, the execution time on modern microprocessors featuring caches and complex pipelines is highly sensitive with respect to the execution history [HLTW03, Rei08]. Last but not least, estimating the WCET based on high-level source code cannot give reliable results if the source

```
l1   int i = x = y = 0;
l2   while (i < 10) {
l3      if (0 == i % 2) {
l4         x++;
        } else {
l5         y++;
        }
l6      i++;
     }
l7   return x – y;
```

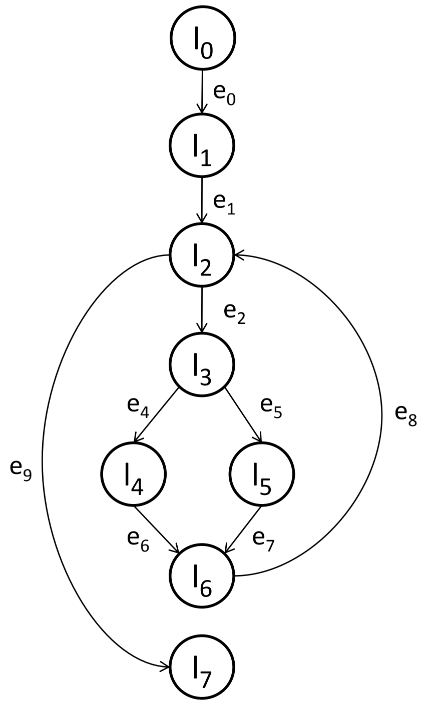


**Figure 2.1.:** Example program and graph with ILP variables.

code is translated by an optimizing compiler. Such compilers apply transformations that preserve the semantics but usually decrease (or at least alter) the execution time. For example, conditionals over a variable that can be proven to be constant for any execution may be removed and loop invariant computations may be hoisted outside the loop body.

There have been attempts to overcome the limitations of Shaw's original method, applying it to assembly level programs and taking the effects of modern processor features into account. To this end, Lim et al. [RLP$^+$94, LBJ$^+$94] and Hur et al. [HBL$^+$95] propose a method which models cache effects via bookkeeping of first and last references to memory blocks; pipelining effects are modeled by reservation tables whose resources are registers and pipeline stages. The approach has been evaluated for the rather simple MIPS 3000 architecture and very small programs. It is limited to such simple targets because the pipeline modeling relies on assumptions that are not valid for more advanced pipelines.

Li et al. propose the use of integer linear programming (ILP) for the static computation of WCET bounds [LMW96]. Their method captures pipelining and cache effects, as well as the task's control flow. The proposed control flow modeling has proven to be very useful for static WCET analysis, hence we discuss it in more detail. The control flow of the analyzed task is modeled as constraints of an ILP. The WCET bound of the task is computed by solving the ILP with a target function that maximizes the execution time. An example of the control flow modeling is depicted in Figure 2.1. The control flow graph on the right-hand side of the figure is decorated with variables for constructing the ILP. The variables $l_1, \ldots, l_7$ denote the execution counts of the labeled instructions in the program and $l_0$ is the execution count of the program entry. Similarly, the variables $e_0, \ldots, e_9$ represent the traversal counts of the corresponding control flow edges. Based on the control flow graph of Figure 2.1, the following ILP constraints can be established:

$$\begin{array}{llll}
l_0 = 1 & l_2 = e_1 + e_8 & l_4 = e_4 & l_6 = e_6 + e_7 \\
l_0 = e_0 & l_2 = e_2 + e_9 & l_4 = e_6 & l_6 = e_8 \\
l_1 = e_0 & l_3 = e_2 & l_5 = e_5 & l_7 = e_9 \\
l_1 = e_1 & l_3 = e_4 + e_5 & l_5 = e_7 & 0 \leq e_2 \leq 10
\end{array}$$

The constraints state that the execution count of an instruction equals the sum of the traversal counts over all incoming and outgoing edges, e.g., $l_2 = e_1 + e_8$ and $l_2 = e_2 + e_9$. Furthermore, that the loop entry at $l_2$ is executed at most ten times, $0 \leq e_2 \leq 10$, and that the whole program is executed only once, $l_0 = 1$. Let $t_i$ denote the execution time for instruction $l_i$. The WCET bound is obtained by solving the ILP under the above constraints, maximizing the following target function, where $t$ denotes the execution time of the whole program:

$$t = \sum_{i=1}^{7} l_i \cdot t_i$$

This technique for bounding the global execution time on the control flow graph is known as *implicit path enumeration technique* or IPET for short. It is also used in the modular, static WCET analysis framework presented in Section 5.1.
Li et al. derive the time bounds for the instructions from another ILP which models the pipeline and cache behavior. The size of this ILP depends on the complexity of the considered architecture. The published analysis times for the rather simple Intel i960KB [Cor91] processor indicate that the approach does probably not scale to more complex architectures.
Healy et al. propose a WCET analysis which uses several distinct phases [HWH95, HAM+99, AMWH94]. The first phase uses a so-called static instruction cache simulator in order to classify instruction fetches as cache hits or misses. A subsequent pipeline path analysis computes execution times for instruction sequences under consideration of the cache simulator results. The pipeline behavior is described in terms of resource usage patterns that are assigned to individual instructions. Finally, the computed execution times are combined into a WCET bound for the analyzed program. Healy et al. consider the MicroSPARC pipeline and direct-mapped caches. Again, the approach is limited to such simple microprocessors because the pipeline modeling relies on assumptions which do not hold for more complex architectures.
Lundquist and Stenström obtain WCET bounds by an interpretation of the program under analysis using an abstract processor simulator [LS98, LS99b]. In contrast to

measurements or explicit simulation, the problem of covering the possible inputs is handled by allowing input ranges for the simulation. The implementation simulates the possible executions by a cycle-wise evolution of the pipeline model and performs ALU operations on ranges instead of concrete values. The WCET bound is derived from the required number of single-cycle simulator steps. Abstract simulation can give safe and tight WCET bounds if the simulation terminates. Also, the method is fully automatic. However, termination is not guaranteed and the complexity of the abstract simulation depends heavily on the number of paths that have to be followed. For example, if the outcome of a branch condition (branch taken or not taken) cannot be precisely predicted, the simulation must continue for both possibilities. The arising complexity can quickly render the analysis infeasible in practice. Lundquist and Stenström evaluate their approach using a hardware model that is based on the PowerPC architecture and rather small programs.

Engblom presents a similar approach in [Eng02]. In contrast to Lundquist and Stenström's abstract simulation, which handles everything in one monolithic analysis, it decomposes the problem into several sub-problems which are solved by separate analyses. However, at the core of the method is also an abstract simulator. The scalability problems of Lundquist and Stenström's abstract simulator are avoided by using the simulation only to compute execution times for linear sequences of instructions. The WCET estimates for all sequences are then combined into a WCET estimate for the whole program. Due to several simplifying assumptions, the approach is limited to simple architectures that are free of timing anomalies (see Section 5.3.2) and do not feature caches, e.g., the ARM7 microprocessor.

Another method for WCET analysis is proposed by Colin and Puaut [CP01]. They call their approach tree-based because it uses the syntax tree of the program under analysis to combine low-level WCET estimates to a global WCET bound. The low-level WCET estimates are obtained from independent analyses of the instruction cache and pipeline behavior. The approach has also been extended to handle dynamic branch prediction [CP00]. A weakness of the tree-based method is that it is not context-sensitive and therefore seems to produce rather pessimistic WCET estimates.

Li/Mitra et al. present an analysis for an out-of-order pipeline which determines context-sensitive execution time bounds for basic blocks by a fixed point analysis of the time intervals at which instructions enter and leave the different pipeline stages [LRM04, LMR05]. The instruction cache and dynamic branch prediction behavior is analyzed by ILP-based techniques [LMR05, RLM02]. The global WCET bound is calculated using IPET. The tool only supports the processor model of the SimpleScalar [ALE02] `sim-outorder` cycle-accurate processor simulator.

Kirner et al. describe a WCET analysis which cooperates with a compiler and requires users to write their programs in a dedicated derivative of the C programming language with WCET annotations [KLFP02]. Execution time bounds are computed on the generated object code using ILP techniques [PS97]. Annotations are taken into account using additional compiler information, particularly in order to deal with compiler optimization [KP03]. The tool supports several simple embedded processors which are free of timing anomalies.

Wilhelm et al. use an approach that decomposes the WCET problem into several sub-problems which are independently solved using a combination of abstract interpretation and integer linear programming [FHL$^+$01, HLTW03]. The implementation starts from fully linked executables and also features analyses for predicting low-level timing effects of the hardware [Fer97, The04]. The analysis has been applied to industrial software [TSH$^+$03, WEE$^+$08] and the analyzer is available as a commercial product [Abs00]. Our work builds on top of this approach which is therefore discussed in detail in Section 5.1.
The research in static WCET analysis led to the development of numerous academic and commercial tools which we have not covered exhaustively, e.g., the Bound-T tool [Tid00, HLS00], or the OTAWA WCET analysis framework [BCRS10]. However, these remaining tools generally use a combination of the already mentioned techniques and deviate only in details. Again, we refer to [WEE$^+$08] for an in-depth discussion.

## 2.3. Symbolic Methods in WCET Analysis

Symbolic methods so far have only been applied to WCET analysis in the context of symbolic model checking. Early work by Logothetis and Schneider computes WCET bounds for synchronous programs, assuming a single-cycle execution model [LS03, LSM03]. This execution model does not hold for modern processor hardware, using pipelined execution and caches to improve the average execution time. It can therefore not be directly compared with any of the already described approaches except timing schemes.
More recent research in model checking for WCET analysis by Larsen et al. also takes the timing effects of the hardware into account [DOT$^+$10]. Pipeline, caches, and the control flow of the analyzed program are modeled as timed automata in the real-time model checker Uppaal [BLL$^+$96, LPY97]. WCET bounds in terms of execution cycles are determined directly by a reachability analysis with the Uppaal model checker. Despite the modular modeling concept, the analysis itself is monolithic in the way that the model checker considers the combined state space of the hard- and software model. It is therefore not surprising that – according to the published results – the approach suffers from serious scalability problems even though it only addresses the most simple pipelines. The work of Larsen et al. is probably the closest to our approach. The similarities concern the modeling of the hard- and software interaction and the fact that symbolic methods are used to cover the state space of the pipeline. But there are also significant differences. Larsen et al. strive for a wholesome model checking approach whereas our work uses BDD-based symbolic methods only for covering the state space of the pipeline. Our approach is also truly modular and therefore less prone to scalability issues.
Multicore processors have recently become a major topic in the WCET research community. The interference between several tasks accessing shared resources significantly complicates the WCET problem. Currently the problem appears far from being solved and it has even been argued that generic multicore processors

should not be considered for applications with WCET requirements [CFG$^+$10]. However, the question of giving WCET bounds for multicore processors also brought forth a research paper on integrating abstract interpretation with model checking [LNYY10]. Wang et al. use an abstract cache analysis to predict the local cache behavior of each task. The interaction of the tasks on the shared bus is then modeled in Uppaal using timed automata. The problem of giving WCET bounds for multicore applications is not quite comparable to the computation of WCET bounds for single uninterrupted tasks. However, it is still interesting to compare the work of Wang et al. with our approach because it also combines abstract interpretation with symbolic state traversal. The difference is that our state traversal is not quite the same as reachability analysis by a symbolic model checker. It is tightly integrated into a static program analysis framework and allows the interaction with other analyses during analysis time (see Section 6.5 for an example).
Finally, a very different approach for using model checking in WCET analysis is taken by Wenzel et al. [WKRP08]. Their tool uses a model checker for finding feasible paths and generating the corresponding input data for measurement-based WCET estimation.

// CHAPTER 3

# Abstract Interpretation

Static program analyses answer questions about the behavior of a computer program without executing it. Their application ranges from questions about the state of variables (*what are the possible values of variable x at program point p*) to more difficult questions about correctness (*does the program never divide by zero?*). A common property of all static analyses is that they compute invariants, i.e., the given answers hold for all possible executions of the program. For any non-trivial analysis, this requires the program semantics to be taken into account. Unfortunately, all non-trivial questions about the concrete semantics of a program are undecidable. Static analyses therefore rely on *abstract semantics* which over-approximates the concrete semantics. Abstractions yield problems that are easier to compute but this is typically associated with a loss of precision. The trade-off between computability and precision makes the design of static analyses a challenging task.

A classic application of static analyses is the design of compiler optimizations. Compiler optimizations involve program transformations that must preserve the program semantics. This requires knowledge about semantic invariants which are obtained by static analyses. More recently, static analysis techniques are also employed in the verification of safety-critical systems, e.g., to prove the absence of run-time errors [BCC$^+$03] and to compute safe upper bounds on the worst-case execution time of tasks [FHL$^+$01].

This chapter gives an introduction into the design of abstract domains for static analyses, the actual computation of the semantic invariants, and into the underlying theory of abstract interpretation. The presentation is based on the works of Miné [Min04], Martin [Mar98, Mar99], and a text book on program analysis by Nielson, Nielson, and Hankin [NNH99].

## 3.1. Lattice Theory

The theory of *abstract interpretation* states that all kinds of semantics can be expressed as fixed points of monotonic functions in partially ordered structures. The correctness of a program analysis can be proved if the considered partially ordered structures satisfy certain conditions. This section introduces the required elements from lattice theory.

**Definition 3.1 (Partially ordered set, poset)**
*A partially ordered set ($\mathcal{D}$,$\sqsubseteq$) is a non-empty set $\mathcal{D}$ with a partial order $\sqsubseteq$, that is a reflexive, transitive and anti-symmetric binary relation. If they exist, the greatest lower bound (glb) of a set $D \subseteq \mathcal{D}$ will be denoted by $\bigsqcap D$ and the least upper bound (lub) will be denoted by $\bigsqcup D$. The least element and greatest element of D will be denoted by $\bot$ and $\top$ if they exist.*

**Definition 3.2 (Complete lattice)**
*A lattice $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap)$ is a poset where each pair of elements $a, b \in \mathcal{D}$ has a least upper bound $a \sqcup b$ and a greatest lower bound $a \sqcap b$. A lattice is said to be complete if any set $D \subseteq \mathcal{D}$ has a least upper bound. This implies that a complete lattice has both a least element $\bot = \bigsqcup \emptyset$ and a greatest element $\top = \bigsqcup \mathcal{D}$. Furthermore, each set $D \subseteq \mathcal{D}$ has a greatest lower bound $\bigsqcap D = \bigsqcup \{X \in D \,|\, \forall Y \in D, X \sqsubseteq Y\}$. A complete lattice will be denoted by $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.*

**Power Set Lattice.** An important complete lattice is the *power set lattice* which can be constructed from any set $S$ using standard set operators, i.e., $(2^S, \subseteq, \cup, \cap, \emptyset, S)$. An example of this lattice based on a set with 4 elements is depicted in Figure 3.1. Its height, i.e., the length of the longest path from $\bot$ to $\top$, is 4. In general, a power set lattice $(2^S, \subseteq, \cup, \cap, \emptyset, S)$ has height $|S|$.

**Abbreviations.** The following text will use a more compact notation for posets and complete lattices if this does not lead to ambiguities. A poset $(\mathcal{D}^x, \sqsubseteq^x)$ will be referred to as $\mathcal{D}^x$, i.e., the superscript $x$ will be used as an identifier for a certain poset. The same holds for lattices, i.e., the complete lattice $\mathcal{D}^x$ is ordered by $\sqsubseteq^x$ with lub $\sqcup^x$, glb $\sqcap^x$, least element $\bot^x$ and greatest element $\top^x$.

We introduce two important properties of functions between lattices. Monotonicity is required for fixed point computation and for establishing a correctness relation between two semantic domains. The notion of distributivity of functions is related to the corresponding property of arithmetic operators.

**Definition 3.3 (Monotonic)**
*A function $f : \mathcal{D}^a \to \mathcal{D}^b$ between two posets $\mathcal{D}^a$ and $\mathcal{D}^b$ is monotonic (or order-preserving) if*

$$\forall X^a, Y^a \in \mathcal{D}^a : X^a \sqsubseteq^a Y^a \Rightarrow f(X^a) \sqsubseteq^b f(Y^a)$$

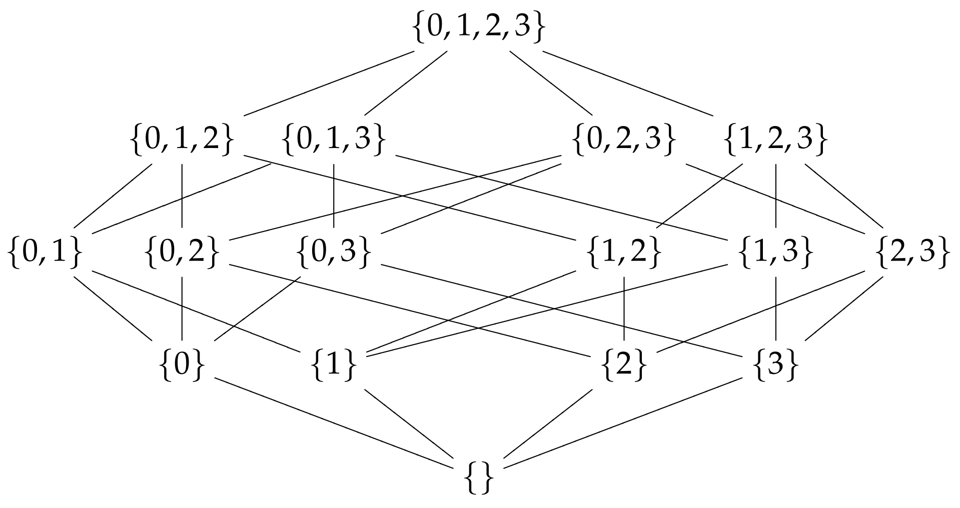


**Figure 3.1.:** Power set lattice for a set with 4 elements.

**Definition 3.4 (Distributive)**
*A function $f : \mathcal{D}^a \to \mathcal{D}^b$ between two lattices $\mathcal{D}^a$ and $\mathcal{D}^b$ is distributive (sometimes called additive) if*

$$\forall X^a, Y^a \in \mathcal{D}^a : f(X^a \sqcup^a Y^a) = f(X^a) \sqcup^b f(Y^a)$$

*It is called completely distributive if*

$$\forall D^a \subseteq \mathcal{D}^a : f(\bigsqcup^a D^a) = \bigsqcup^b \{f(X^a) \mid X^a \in D^a\}$$

*whenever $\bigsqcup^a D^a$ exists.*

## 3.1.1. Fixed Points

A *fixed point* of a function $f$ is an element $X$ such that $f(X) = X$. If $f$ is monotonic, the following theorem ensures the existence of fixed points.

**Theorem 3.1 (Tarski's Fixed Point Theorem)**
*If $f : \mathcal{D}^x \to \mathcal{D}^x$ is a monotonic function in a complete lattice $\mathcal{D}^x$, then the set of all fixed points $Fix(f) = \{X^x \mid f(X^x) = X^x\}$ is nonempty and forms a complete lattice when ordered by $\sqsubseteq^x$.*

**Proof** Found in [Tar55]. ■

The practical computation of fixed points is based on ascending sequences of elements in complete lattices called *ascending chains*.

**Definition 3.5 (Ascending chain)**
*An ascending chain $(X_i)_i$ is a sequence $X_0, X_1, \dots$ such that $\forall j : X_j \sqsubseteq X_{j+1}$. It is called* strictly ascending *iff $\forall j : X_j \neq X_{j+1}$. An ascending chain is said to stabilize if it holds that $\exists j : \forall n \geq j : X_j = X_n$.*

An example of an ascending chain in the power set lattice of Figure 3.1 is the following sequence which, in terms of the picture, ascends along the left edge of the lattice:

$$\bot = \emptyset, \{0\}, \{0,1\}, \{0,1,2\}, \{0,1,2,3\} = \top$$

The non-existence of strictly ascending chains naturally gives rise to an iterative computation of fixed points. This is formalized by the following theorem.

**Theorem 3.2 (Kleene's Fixed Point Theorem)**
*Let $\mathcal{D}^x$ be a complete lattice where all ascending chains eventually stabilize. If $f : \mathcal{D}^x \to \mathcal{D}^x$ is a monotonic function, then there exists a $k$ such that $f^k(\bot^x) = f^{k+1}(\bot^x)$ and $f^k(\bot^x)$ is the least fixed point of $f$.*

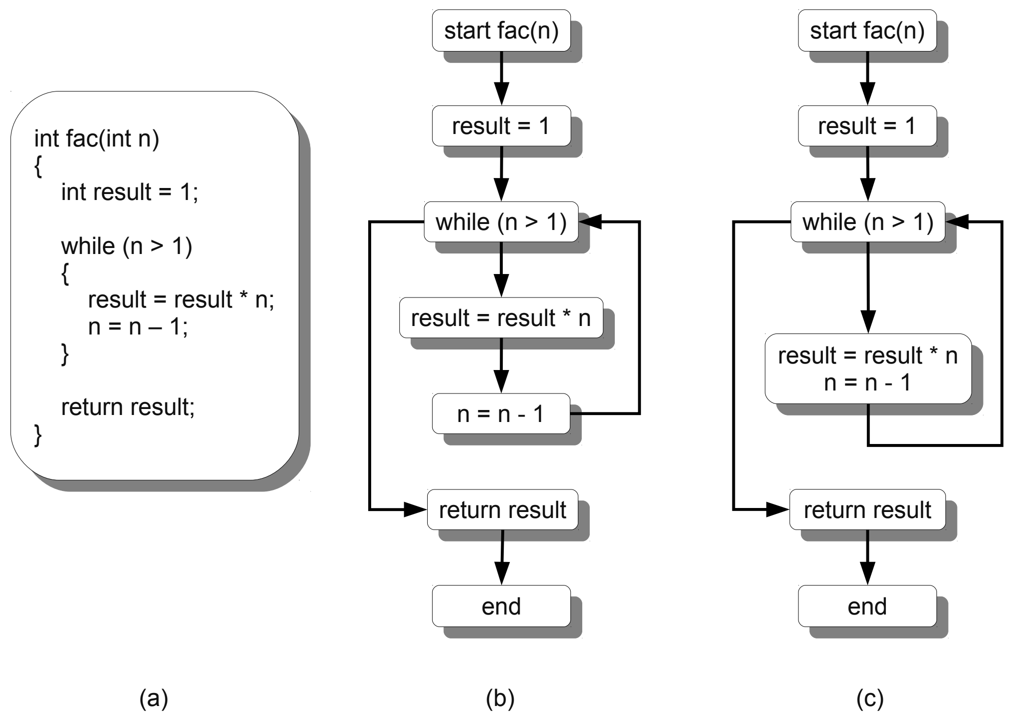
**Proof** Found in [Cou78]. ■

Iterative fixed point computation might not converge if the lattice has strictly ascending chains. In such cases, convergence can be enforced by the application of a *widening* operator. The obtained fixed point is typically above the least fixed point. A subsequent down iteration using a *narrowing* operator can refine such coarse overapproximation. Interval analysis is an example where widening plays an important role. It assigns an interval of natural numbers to integer variables. The top element of this lattice is the interval $[-\infty, +\infty]$. The analysis of a loop incrementing an integer variable $x$ in every loop iteration may not reach a fixed point if it fails to determine a static bound for the number of loop iterations. Widening will force convergence by setting the interval for $x$ to $[-\infty, +\infty]$.
Within the scope of our work, widening and narrowing are not required since we only consider lattices of finite height where convergence speed is no problem. We refer to [NNH99] for details on this subject.

## 3.2. Collecting Semantics

A static program analysis has to refer to the semantics of programs. We consider a form of semantics that is based on a program representation as a directed graph.

**Definition 3.6 (Control flow graph)**
*A control flow graph (CFG) is a directed graph $G = (N, E, s, e)$ with a finite set of nodes $N$, a set of edges $E \subseteq N \times N$, a start node $s$ and an end node $e$. If there exists an edge $(n, m)$*



```
int fac(int n)
{
   int result = 1;

   while (n > 1)
   {
      result = result * n;
      n = n – 1;
   }

   return result;
}
```

**Figure 3.2.:** (a) Source code of a procedure for computing the factorial of an integer number, (b) the procedure's control flow graph, and (c) the basic block graph of the same procedure.

*then $n$ is called predecessor of $m$ and $m$ is called successor of $n$. The start node $s$ has no predecessor and the end node $e$ has no successor.*
*A sequence $(n_1, \ldots, n_k) \in N^*$ is a path through $G$ iff $n_1 = s$ and $\forall j \in \{1, \ldots, k-1\} : (n_j, n_{j+1}) \in E$. A sequence $(n_1, \ldots, n_k) \in N^*$ is a path to $n$ if it is a path and $n_k = n$. The path to the start node $s$ is denoted by $(s) \in N^*$.*

It is often useful to collapse maximal paths of strictly linear control flow into single blocks. This transformation yields a more compact CFG representation in which each node subsumes one or several program statements.

**Definition 3.7 (Basic block)**
*A basic block is a maximal sequence of nodes $(n_1, ..., n_k)$ of a $CFG = (N, E, s, e)$ such that $n_i$ is the only predecessor of $n_{i+1}$ and $n_{i+1}$ is the only successor of $n_i$ and $n_1 \neq s$, $n_k \neq e$. A basic block graph $G = (N, E, s, e)$ is a CFG in which every node $n \in N$ that is not the start or end node, is a basic block.*

Figure 3.2 shows the C source code of a function for computing the factorial of an integer number and its corresponding control flow and basic block graphs.

Path semantics expresses the meaning of program statements in a CFG by a function $f^b : E \to \mathcal{D}^b \to \mathcal{D}^b$ over a domain which is required to be a complete lattice. For a given edge $(m, n) \in E$ we say that $f^b$ computes the effect of the program statement $m$ on a state $x^b \in \mathcal{D}^b$. The effect of the statement $m$ can depend on the target node $n$, e.g., if the semantics captures branch conditions.

**Definition 3.8 (Path semantics)**
*The semantics of a path $\pi = (n_1, \ldots, n_k) \in N^*$ through the CFG $G = (N, E, s, e)$ is defined as*

$$[\![\pi]\!] := \begin{cases} \lambda x.x & \text{if } \pi = (n) \\ [\![n_2, \ldots, n_k]\!] \circ f^b(n_1, n_2) & \text{if } \pi = (n_1, \ldots, n_k), k > 1 \end{cases}$$

Program analysis calculates invariants that hold for all input data. This can be expressed by lifting the path semantics to sets of states. Let $(\mathcal{D}^{coll}, \subseteq, \cup, \cap, \emptyset, \mathcal{D}^b)$ be a power set domain over the elements of $\mathcal{D}^b$. This domain is a complete lattice, independent of the properties of $\mathcal{D}^b$. The effect of executing a program statement in this domain can be represented by a function $f^{coll} : E \to \mathcal{D}^{coll} \to \mathcal{D}^{coll}$.

**Definition 3.9 (Collecting semantics)**
*The collecting semantics of a path $\pi = (n_1, \ldots, n_k) \in N^*$ through the CFG $G = (N, E, s, e)$ is defined as*

$$[\![\pi]\!]_{coll} := \begin{cases} \lambda x.x & \text{if } \pi = (n) \\ [\![n_2, \ldots, n_k]\!]_{coll} \circ f^{coll}(n_1, n_2) & \text{if } \pi = (n_1, \ldots, n_k), k > 1 \end{cases}$$

*The collecting semantics of a node $n \in N$ w.r.t. a set of input states $\iota$ is defined as*

$$Coll_G(n) := \bigcup \{ [\![\pi]\!]_{coll}(\iota) \mid \pi \text{ is a path to } n \text{ in } G \}$$

For an abstract domain with computable semantics, the collecting semantics for all nodes of the CFG of a procedure can be computed by a data flow analysis. It computes the Kleenian fixed point of a monotonic function in the underlying domain in an iterative fashion. A data flow analysis can be defined in terms of

- a data flow *domain* $\mathcal{D}^\sharp$, required to be a complete lattice,
- a monotone *transfer function* $tf^\sharp : E \to \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ for updating domain elements,
- a *least upper bound operator* $\sqcup^\sharp$ for combining domain elements at control flow joins,
- an *(in-) equality operator* $=^\sharp$ or $\neq^\sharp$ for checking for fixed points of the transfer function. Note that $=^\sharp$ and $\neq^\sharp$ can always be constructed from the $\sqsubseteq^\sharp$ operator.

The least fixed point of $tf^\sharp$ is computed by iteration over the nodes of the graph. The corresponding algorithm is depicted in Figure 3.3. The iteration is guided by a workset of nodes and $v(n) \in \mathcal{D}^\sharp$ is the set of values assigned to a node $n \in N$. The set $\iota^\sharp$ of initial values is assigned to the entry point of the analyzed procedure.

```
1: for all $n \in N$ do
2:     $v(n) = \bot^\sharp$
3: end for
4: $v(s) = \iota^\sharp$
5: workset $= \{n \mid (s,n) \in E\}$
6: while workset $\neq \emptyset$ do
7:     let $n \in$ workset in
8:     workset $=$ workset $\setminus \{n\}$
9:     $X = \bigsqcup^\sharp \{tf^\sharp(m,n)(v(m)) \mid (m,n) \in E\}$
10:     if $X \neq^\sharp v(n)$ then
11:         $v(n) = X$
12:         workset $=$ workset $\cup \{m \mid (n,m) \in E\}$
13:     end if
14: end while
```

**Figure 3.3.:** Algorithm for computing the minimum fixed point solution of a data flow problem on the CFG of a procedure.

## 3.2.1. Galois Connections

The semantic invariants that are computed by a program analysis should be correct w.r.t. the considered concrete semantics. For abstract interpretation, this means that the computed fixed points in the employed abstract domain over-approximate the concrete collecting semantics. This relationship between the concrete and abstract semantic domains is formalized by the notion of *Galois connection* [CC77].

**Definition 3.10 (Galois connection)**
*Let $\mathcal{D}^b$ and $\mathcal{D}^\sharp$ be two posets denoting respectively a concrete and an abstract semantic domain. A Galois connection between $\mathcal{D}^b$ and $\mathcal{D}^\sharp$ is a pair of functions $(\alpha, \gamma)$ such that:*

$$\forall X^b, X^\sharp : \alpha(X^b) \sqsubseteq^\sharp X^\sharp \Longleftrightarrow X^b \sqsubseteq^b \gamma(X^\sharp) \tag{3.1}$$

To put it informally, abstracting and concretizing an element from the concrete domain using an abstraction function $\alpha$ and a concretization function $\gamma$ always yields an overapproximation. The existence of a Galois connection ensures *local consistency*. Analysis results in the abstract domain over-approximate the concrete semantics at all nodes in the CFG. Hence, the analysis is correct. The Galois connection between two domains and its relation to the local consistency of the semantics are depicted in Figure 3.4.

**Theorem 3.3**
*If $(\mathcal{D}^b, \alpha, \gamma, \mathcal{D}^\sharp)$ is a Galois connection then $\alpha$ and $\gamma$ are both monotonic.*

$$\begin{tikzcd}
tf^\flat(e)(X^\flat) \sqsubseteq^\flat & & tf^\sharp(e)(X^\sharp) \arrow[ll, "\gamma"'] \\
X^\flat \arrow[u, "tf^\flat(e)"] \arrow[rr, "\alpha"] & & X^\sharp \arrow[u, "tf^\sharp(e)"']
\end{tikzcd}$$

**Figure 3.4.:** Concrete and abstract semantic domains $\mathcal{D}^\flat$ and $\mathcal{D}^\sharp$ that are connected by a Galois connection $(\alpha, \gamma)$. For any edge $e \in E$ of a CFG $G = (N, E, s, e)$, the application of a transfer function $tf^\sharp$ in the abstract domain over-approximates the semantics of the corresponding computation in the concrete domain by $tf^\flat$.

**Proof** For $\gamma$, assume

$$X^\sharp \sqsubseteq^\sharp Y^\sharp \tag{3.2}$$

The following sequence of transformations shows that $\gamma$ is monotonic. The initial equation holds because the partial order operator $\sqsubseteq^\flat$ is reflexive (Definition 3.1).

$$\begin{aligned}
& \gamma(X^\sharp) \sqsubseteq^\flat \gamma(X^\sharp) \\
\overset{3.1}{\Longrightarrow}\; & \alpha(\gamma(X^\sharp)) \sqsubseteq^\sharp X^\sharp \\
\overset{3.2}{\Longrightarrow}\; & \alpha(\gamma(X^\sharp)) \sqsubseteq^\sharp Y^\sharp \\
\overset{3.1}{\Longrightarrow}\; & \gamma(X^\sharp) \sqsubseteq^\flat \gamma(Y^\sharp)
\end{aligned}$$

The proof for $\alpha$ can be easily constructed using the same idea. ∎

To ensure the existence of a Galois connection, it is not necessary to specify both the abstraction and concretization function. If $\alpha$ is monotonic and completely distributive and $\mathcal{D}^\flat$ is a complete lattice, then the missing function can be synthesized in a canonical way as is stated by the following theorem.

**Theorem 3.4**
*If $(\mathcal{D}^\flat, \alpha, \gamma, \mathcal{D}^\sharp)$ is a Galois connection then $\alpha$ is completely distributive and uniquely determines $\gamma$ by*

$$\gamma(X^\sharp) = \bigsqcup^\flat \{X^\flat \mid \alpha(X^\flat) \sqsubseteq^\sharp X^\sharp\}$$

*A dual theorem holds for $\gamma$ with $\bigsqcap$ instead of $\bigsqcup$.*

**Proof** Found in [NNH99]. ∎

## 3.3. Interprocedural Analysis

Programs are usually composed of many functions and procedures. Each procedure can be called from different *call contexts*. To arrive at a precise analysis, the different call contexts have to be analyzed separately. Martin [Mar98] lists several solutions for tackling this problem:

- *Inlining:* Procedure calls are replaced by the body of the callee. This is only feasible for non-recursive procedures. Further, the program representation grows significantly.
- *Effect calculation:* Every execution of a procedure computes an *effect*. This is a function which maps the input values of the procedure to its outputs.
- *Call string:* The call history is incorporated into the analysis domain and used to distinguish domain elements in different call contexts.
- *Static call graph:* Call sequences are statically computed prior to the data flow analysis. Each sequence is then analyzed separately.

The call string and static call graph approaches allow the analyzer to lose precision by merging call histories. For call strings, this is achieved by cutting off call strings whose lengths exceed a certain threshold. Similarly, the computation of call sequences in a static call graph can be limited, and sequences that exceed the threshold are merged.
The merging of call histories allows to configure the precision and computational complexity of an interprocedural analysis. Distinguishing longer call histories yields more precise analyses, but requires more memory and computation time. The most precise analysis without merging call histories is often infeasible in practice.
We consider the static call graph approach which is used by the program analyzer generator PAG [Mar98]. It is based on a special CFG, called *supergraph*, which interconnects the basic block graphs of individual procedures. Local edges [RHS95] allow for the propagation of information that is invariant with regard to procedure calls. Interprocedural analysis problems can be reduced to intraprocedural analysis problems on the supergraph.

**Definition 3.11 (Supergraph)**
*Let $G_1, \ldots, G_n$ be the CFGs of all n procedures of a program P. The supergraph $G^* = (N^*, E^*, s^*, e^*)$ of P consists of a set of nodes $N^*$ that is the union of all node sets of $G_1, \ldots, G_n$ except that every node representing a procedure call from a procedure $P_i$ to a procedure $P_j$ is replaced by two nodes:*

- *a call node $n^c_{i,j}$,*
- *a return node $n^r_{i,j}$.*

*The set of edges $E^*$ contains all edges of $G_1, \ldots, G_n$, and the following additional edges for every procedure call from a procedure $P_i$ to a procedure $P_j$:*

- *a call edge from the call node $n^c_{i,j}$ in $P_i$ to the start node $s_j$ of $P_j$,*
- *a return edge from the exit node $e_j$ of $P_j$ to the return node $n^r_{i,j}$ in $P_i$,*
- *a local edge from the call node $n^c_{i,j}$ to the return node $n^r_{i,j}$.*

*The start and end nodes of $G^*$ are the start and end nodes of the entry procedure $G_1$: $s^* = s_1$, $e^* = e_1$.*

The implementation of the static call graph approach on the supergraph works as follows. Each node in the supergraph is annotated with an array of elements from the analysis domain. The number of data elements at nodes of the same procedure is fixed. It corresponds to the number of different call contexts in which the procedure is analyzed. Corresponding array cells between two neighboring nodes in the supergraph are connected by a set of connector functions. To merge call histories between two nodes, the connector functions connect several array slots at the predecessor node with a single array slot at the successor node. The resulting graph is called *extended supergraph*. An example of this graph is depicted in Figure 3.5.
The extended supergraph allows the analysis of recursive functions with unknown recursion bounds by merging recursive calls that exceed a chosen threshold. It can also be used for precise loop analysis. To this end, loops are transformed into recursive procedures [MAWF98]. *Virtual loop unrolling* refers to the assignment of a static number of distinguished analysis contexts to a recursive loop procedure.
The analyzer generator PAG [Mar98] uses the described implementation of the static call graph approach. Context handling is referred to as *virtual inlining, virtual unrolling*, or VIVU for short [MAWF98].

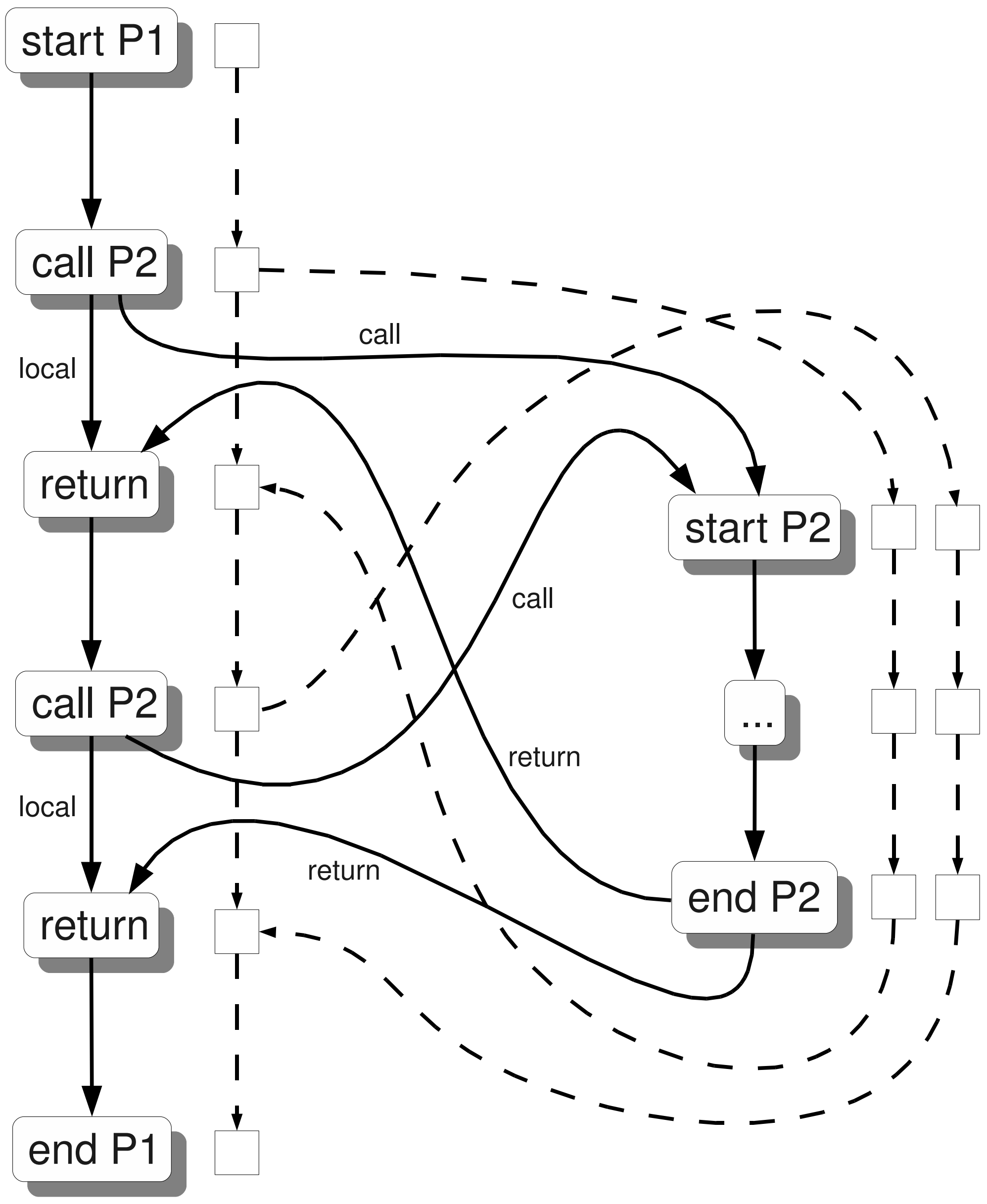


**Figure 3.5.:** Extended supergraph with two procedures P1 and P2. P1 calls P2 twice which is analyzed in two contexts. The rounded boxes are nodes of the supergraph and the small square boxes represent the static context arrays. The solid edges are control flow edges of the supergraph whereas the dashed edges connect the static contexts.

# CHAPTER 4

# Symbolic State Space Exploration

Modern computer hardware can be regarded as a complex network of digital circuits which are composed of gates and interconnecting wires. A gate is a collection of transistors that outputs the result of a Boolean operation on its inputs. Memory elements such as binary latches provide the additional capability of storing signals for further use in later computations. The behavior of digital circuits can be precisely captured by *switching functions* and *finite state machines*. Switching functions correspond to stateless circuits (also called *combinatorial circuits*) whereas *finite state machines* (FSMs) provide a model for *sequential circuits* in which the result of a computation also depends on a memory state. Both computation models, switching functions and finite state machines, play an important role in the design and formal verification of digital circuits.

Switching functions and finite state machines can be represented by different data structures, like Boolean formulae, truth tables, and decision diagrams. Choosing a compact representation is of particular importance for formal verification. Model checking techniques, which are widespread in hardware verification, require a representation of the transition system to be verified and often traverse a substantial part of its state space. The explicit construction of large transition systems is infeasible because the number of states is exponential in the number of memory elements. This problem is known as the *state explosion* problem. It also affects the representation of analysis information, i.e., sets of reachable states, during the state traversal. Symbolic representations, in particular *ordered binary decision diagrams* (BDDs), avoid the explicit enumeration of states. They admit an *implicit* encoding of the transition system and of analysis information, like sets of states in state traversal. This approach has enabled the verification of large circuits by model checking [BCM+90].

This chapter gives an introduction into the representation of FSM transition systems and sets of states using Boolean functions. Ordered binary decision diagrams are introduced as a data structure for the compact representation of Boolean functions.

Finally, we discuss a method for the implicit state traversal of FSMs using a symbolic representation based on BDDs. The presentation is based on a text book about VLSI[1] design by Meinel and Theobald [MT98], the original paper on BDDs by Bryant [Bry86], and a paper on image computation by Ranjan et al. [RAB+95].

## 4.1. Switching Functions

The switching algebra is a Boolean algebra with two elements. It precisely captures the behavior of combinatorial circuits and is therefore commonly regarded as the theoretical foundation of circuit design.

**Definition 4.1 (Switching algebra)**
*Let the operations $+, \cdot, ^{-}$ on the set $\mathbb{B} = \{0, 1\}$ be defined as follows:*

- $a + b = max\{a, b\}$
- $a \cdot b = min\{a, b\}$
- $\overline{0} = 1$ *and* $\overline{1} = 0$

*Then $(\mathbb{B}, +, \cdot, ^{-})$ is a Boolean algebra called the switching algebra.*

**Definition 4.2 (Switching function)**
*An n-variable function $f : \mathbb{B}^n \to \mathbb{B}$ is called a switching function. The set of all n-variable switching functions is denoted by $\mathbb{B}_n$.*

A basic operation for computations with switching functions is Shannon's expansion which establishes a relationship between a function and its subfunctions. Subfunctions are derived from a function by assigning constants to some of its input variables.

**Theorem 4.1 (Shannon expansion)**
*Let $f \in \mathbb{B}_n$ be a switching function. For the subfunctions $g, h \in \mathbb{B}_{n-1}$ defined by*

$$g(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, 0)$$
$$h(x_1, \ldots, x_{n-1}) = f(x_1, \ldots, x_{n-1}, 1)$$

*it holds that*

$$f = \overline{x_n} \cdot g + x_n \cdot h$$

[1] VLSI stands for Very Large Scale Integration.

**Proof** Let $a_1, \ldots, a_n$ be an assignment to the input variables $x_1, \ldots, x_n$. For $a_n = 0$ the theorem follows from the equation $f(a_1, \ldots, a_n) = g(a_1, \ldots, a_{n-1})$. For $a_n = 1$ it follows from the equation $f(a_1, \ldots, a_n) = h(a_1, \ldots, a_{n-1})$. ∎

Theorem 4.1 derives subfunctions by fixing the last input variable of a function. The same idea can also be applied to other subfunctions and to other operations. The Shannon expansion with respect to the $i$-th argument is given by:

$$f(x_1, \ldots, x_n) = x_i \cdot f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) + \overline{x_i} \cdot f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$$

The dual of Shannon's expansion is given by:

$$f(x_1, \ldots, x_n) = (\overline{x_i} + f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)) \cdot (x_i + f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n))$$

**Note:** *Each n-variable switching function is also a Boolean function. In the following, the terms Boolean function and switching function will therefore be used synonymously.*

## 4.2. Ordered Binary Decision Diagrams

Ordered binary decision diagrams (BDDs) provide a canonical representation of switching functions that is very compact for many switching functions of practical relevance. Further, Boolean operations and equivalence checks can be carried out efficiently.

**Definition 4.3 (Ordered binary decision diagram, BDD)**
*Let $<$ be a total order on the set of variables $x_1, \ldots, x_n$. An ordered binary decision diagram is a directed acyclic graph with a single root node which satisfies the following properties:*

- *There are two terminal nodes labeled by the constants 1 and 0.*
- *Each non-terminal node is labeled by a variable $x_i$ and has two outgoing edges that are labeled by 0 and 1, respectively.*
- *For each edge leading from a node labeled by $x_i$ to a node labeled by $x_j$ it holds that $x_i < x_j$. Consequently, on any path in the graph, variables appear in the order defined by $<$.*

*The variable labeling a node $v$ is denoted by $var(v)$. $low(v)$ and $high(v)$ denote the successor node reached by the 0-edge or 1-edge, respectively.*

An input $a = (a_1, \ldots, a_n) \in \mathbb{B}^n$ assigning values to all variables $x_1, \ldots, x_n$ in a BDD defines a path through the graph. The path begins at the root node and at a node labeled $x_i$ it follows the edge labeled by $a_i$. This establishes a direct correlation between switching functions and BDDs.

**Definition 4.4 (Representation of switching functions)**
*A BDD represents a switching function $f \in \mathbb{B}_n$ if for all inputs $a \in \mathbb{B}^n$ the path defined by a leads to the terminal node with label $f(a)$.*

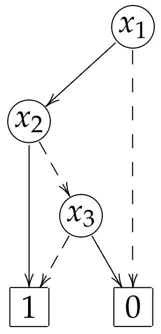


**Figure 4.1.:** BDD for the switching function $x_1 \cdot x_2 + x_1 \cdot \overline{x_2} \cdot \overline{x_3}$ with variable ordering $x_1, x_2, x_3$. Solid arrows represents 1-edges and dashed arrows represent 0-edges of non-terminal nodes.

Assigning a value to a BDD node defines a Shannon expansion of the corresponding switching function. In BDD terminology, the subfunctions obtained by Shannon's expansion are called *cofactors*.

**Definition 4.5 (Cofactor)**
*Let $f \in B_n$ be a switching function. The positive cofactor of $f$ with respect to the i-th input $x_i$ is the subfunction $f_{x_i} = f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n)$. The negative cofactor of $f$ with respect to $x_i$ is the subfunction $f_{\overline{x_i}} = f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n)$.*

If the root node of a BDD for a switching function $f$ is labeled by the variable $x_i$, Shannon's expansion on that variable will be written as

$$f = x_i \cdot f_{x_i} + \overline{x_i} \cdot f_{\overline{x_i}} \tag{4.1}$$

## 4.2.1. Isomorphism and Reduction

BDDs according to Definition 4.3 may contain redundant information. The definition of redundancy in BDDs is based on the notion of isomorphism.

**Definition 4.6 (Isomorphism of BDDs)**
*Two BDDs A and B are called isomorphic if there is a bijective mapping $\phi$ from the nodes of A to the nodes of B such that for each node v one of the following propositions hold:*

1. *$v$ and $\phi(v)$ are terminal nodes with identical labels, or*
2. *$var(v) = var(\phi(v))$ and $\phi(high(v)) = high(\phi(v))$ and $\phi(low(v)) = low(\phi(v))$.*

As a consequence of redundancy, the same switching function can be represented by different BDDs. A canonical data structure is more desirable because it allows for easier checking of equivalence. *Reduced BDDs* provide such a data structure.

**Definition 4.7 (Reduced BDD)**
*A BDD is called reduced if*

1. *it does not contain a node $v$ where $high(v) = low(v)$, and*
2. *there is no pair of nodes $u$, $v$ such that the BDDs rooted in $u$ and $v$ are isomorphic.*

The two constraints on BDDs imposed by the above definition imply two reduction rules which define how any BDD can be transformed into a reduced BDD.

- **Elimination rule:** If it holds for a node $v$ that $u = high(v) = low(v)$ then $v$ can be removed from the graph. All incoming edges of $v$ are redirected to $u$.
- **Merging rule:** If for two nodes $u$ and $v$ it holds that $var(u) = var(v)$ and $high(u) = high(v)$ and $low(u) = low(v)$, then one of the two nodes can be eliminated from the graph. Incoming edges of the eliminated node are redirected to the remaining node.

**Theorem 4.2**
*A BDD is reduced if neither of the two reduction rules can be applied.*

**Proof** Found in [MT98]. ■

For a BBD with $n$ nodes the reduction algorithm that is based on these rules has a time complexity that is bounded by $O(n \cdot \log n)$ [MT98]. Reduced BDDs provide a canonical and minimal representation of switching functions. This means that for a fixed variable order, the reduced BDD of a switching function is uniquely determined up to isomorphisms as stated by the following theorem:

**Theorem 4.3 (Canonical Representation)**
*For any switching function $f$, there is a unique (up to isomorphism) reduced BDD denoting $f$ and any other BDD denoting $f$ contains more vertices.*

**Proof** Found in [Bry86]. ■

Because of the correspondence between reduced BDDs and switching functions, we say that a path that starts at the root node and ends at the terminal node 1 is a *satisfying path*; it corresponds to one or several satisfying assignments to the inputs of the corresponding switching function. Observe that not all variables need to appear on all paths. For example, the BDD of Figure 4.1 has four paths; only two paths comprise nodes for all three variables. Variables that do not appear on a path are sometimes attributed to so-called *dont-care nodes*. These nodes can be omitted because they do not influence the result.

### 4.2.2. Binary Operations and Equivalence Test

We briefly discuss how Boolean operations and equivalence checking on two BDDs can be carried out. As an example of a Boolean operation, we consider the conjunction of two functions $f$ and $g$. The computation of binary operations on BDDs is based on Shannon's expansion with respect to the leading variable in the variable order of the involved BDDs.[2] Let $x_i$ be that leading variable. By application of Equation 4.1 we obtain the following equivalence:

$$f \cdot g = x_i \cdot (f_{x_i} \cdot g_{x_i}) + \overline{x_i} \cdot (f_{\overline{x_i}} \cdot g_{\overline{x_i}}) \tag{4.2}$$

A BDD for $f \cdot g$ can be constructed from the functions $(f_{x_i} \cdot g_{x_i})$ and $(f_{\overline{x_i}} \cdot g_{\overline{x_i}})$ by introducing a new node labeled $x_i$ and by setting $high(x_i)$ to the root node of $(f_{x_i} \cdot g_{x_i})$ and $low(x_i)$ to the root node of $(f_{\overline{x_i}} \cdot g_{\overline{x_i}})$. The functions $(f_{x_i} \cdot g_{x_i})$ and $(f_{\overline{x_i}} \cdot g_{\overline{x_i}})$ can be constructed recursively using the same idea.
A straightforward implementation of this recursive computation would require $2^n$ recursive calls for $n$ being the number of variables. Efficient implementations make use of the following observations:

1. Each recursive call in the computation has two arguments $f'$ and $g'$.
2. In each call, $f'$ is a subfunction of $f$ and $g'$ is a subfunction of $g$.
3. Each subfunction of $f$ and $g$ corresponds to exactly one BDD node.

Multiple calls with the same pair of arguments can be avoided by caching already computed results. This dramatically reduces the number of required recursive calls. It is then bounded by the product of the number of nodes of both BDDs. The same recursive implementation with caching of intermediate results can also be used for other binary operations. The result is in general not a reduced BDD. A reduced BDD can be obtained by application of the reduction rules.
The described binary operations can be used for the construction of BDDs for complex Boolean formulae starting from primitive BDDs. Figure 4.2 gives an example that illustrates the construction of the BDD of Figure 4.1. This construction method is convenient when using a BDD library, for example CUDD [Som09], for computing the binary operations. Other approaches for manual BDD construction can be found in [MT98].

**Equivalence Test.** For testing the equivalence of two reduced BDDs it suffices to check whether the two BDDs are isomorphic. This is guaranteed by Theorem 4.3. The equivalence test can be implemented by simultaneous traversal of both BDDs using depth first search. For each visited pair of nodes, both nodes have the same label. If this is true for all nodes, then both BDDs represent the same function.

**Note:** *In the following, we assume that all BDDs are reduced, i.e., the term BDD always means* reduced *BDD.*

[2]Note that both BDDs must rely on exactly the same variable order.

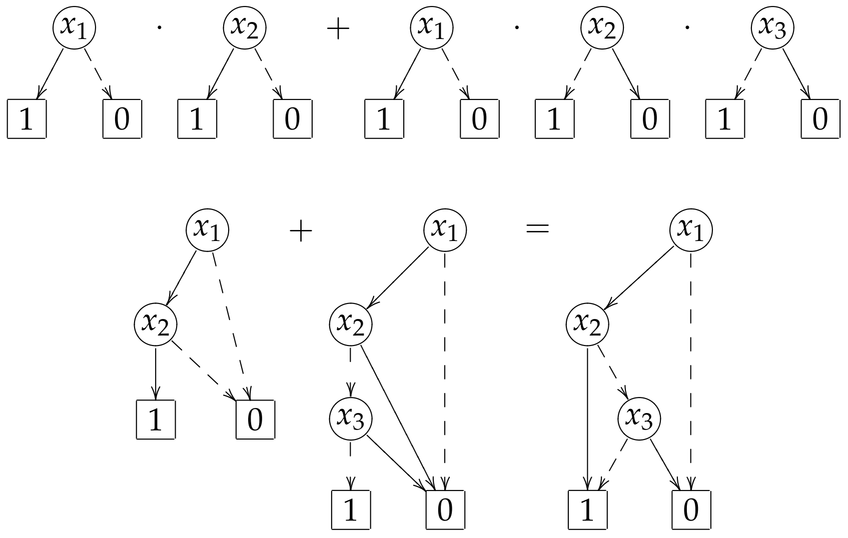


**Figure 4.2.:** Construction of the BDD of Figure 4.1 using binary BDD operations. The conjunctions are computed by recursive application of Equation 4.2. The disjunction is handled equivalently. Note that all BDDs in this example have already been reduced by application of the elimination and merging rules.

### 4.2.3. Influence of the Variable Order

The size of BDDs and thus the complexity of BDD operations depends not only on the number of inputs of the corresponding switching function, but also on the chosen variable order. This influence can have a significant impact as is demonstrated by the example of Figure 4.3. The two depicted BDDs for the function $x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ differ only in the underlying ordering of variables. The first ordering yields a graph with 8 nodes whereas the graph obtained with the second ordering has 16 nodes. In general, for functions of the form

$$f(x_1, \ldots, x_n) = x_1 \cdot x_2 + x_3 \cdot x_4 + \ldots + x_{2n-1} \cdot x_{2n}$$

the BDD size for the variable order $x_1, x_2, \ldots, x_{2n}$ is exactly $2n + 2$. In contrast, for the variable order $x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, x_{2n}$ the BDD size is $2^{n+1}$. This means that choosing a bad variable order can turn a linear representation into an exponential one. Since the complexity of BDD operations depends on the size of the involved BDDs, it is important to obtain a BDD of minimal size. Unfortunately, the test whether a BDD of a function is minimal is **NP**-complete. Therefore, constructing a minimal BDD for a given function is an **NP**-hard problem [THY93].

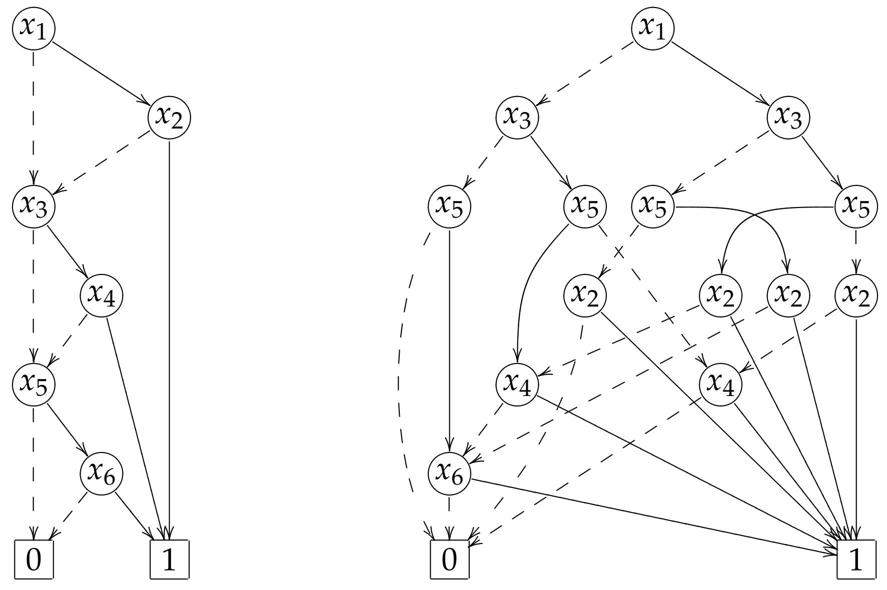


**Figure 4.3.:** Two BDDs for the same switching function $x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$ but with different variable orders. The left-hand graph is based on the order $x_1, x_2, x_3, x_4, x_5, x_6$. The right-hand graph is based on the order $x_1, x_3, x_5, x_2, x_4, x_6$. This example was first published in [Bry86].

## 4.2.4. Dynamic Reordering

Various heuristics have been invented for minimizing the size of BDDs in practice. Some of these heuristics are specific to certain types of BDDs arising only in specific computations. Usually they cannot be successfully applied to other types of BDDs. A general class of heuristic optimizations is based on a *dynamic reordering* of variables. Dynamic reordering algorithms swap variables in the variable order and evaluate the effect of different permutations on the BDD size. The most important dynamic reordering algorithms are *window permutation* [FMK91, ISY91] and *sifting* [Rud93]. The sifting algorithm is often considered to be the most effective reordering algorithm.

A particular advantage of dynamic reordering strategies is that they can be used in an automated fashion without user interaction. Common BDD libraries, e.g., CUDD [Som09], use a *shared* BDD representation. That is, subgraphs which occur in several BDDs are only created once and then shared among these BDDs. The automatic reordering algorithm is invoked whenever the size of the shared BDD representation exceeds a certain threshold. After each reordering, the threshold is increased in order to prevent too frequent reorderings.

## 4.3. Sequential Circuits

Sequential circuits, such as microprocessors, are composed of the following basic elements:

- *Gates* are collections of transistors that output the result of Boolean operations on their inputs.
- *Wires* transmit binary signals between gates and memory elements.
- *Latches* are binary memory elements that store signals for further use in later computations. Latches in synchronous systems are *clocked*, i.e., they are only updated at the edges of a clock signal. Such latches are also known as *flip-flops*.

In addition to these basic elements, sequential circuits may also have *inputs* and *outputs*, i.e., signals that are received from or sent to external devices. Complex sequential circuits are usually specified in high-level languages such as VHDL or Verilog. These specifications can be translated into a low-level description called a *netlist*. The netlist describes the circuit directly in terms of its latches, logic gates and wires and can be used to generate the hardware layout. Circuit verification, e.g., by model checking, can also be based on such low-level representations. We now establish the connection between sequential circuits and finite state machines.

**Definition 4.8 (Finite state machine)**
*A (deterministic) finite state machine $(Q, I, O, \delta, \lambda, q_0)$ consists of a set of states $Q$, an input alphabet $I$, an output alphabet $O$, a next-state function $\delta : Q \times I \rightarrow Q$, an output function $\lambda : Q \times I \rightarrow O$ and an initial state $q_0$.*

A sequential circuit with $n$ latches, $m$ output wires and $t$ input wires is characterized by an FSM with state space $Q = \mathbb{B}^n$, output space $O = \mathbb{B}^m$ and input space $I = \mathbb{B}^t$. The next state and output functions are defined by the corresponding logic of the circuit. State transitions defined by the next state function are associated to clock ticks. Note that the next state function can be constructed individually for each latch. For a latch $k$ we write its next state function as: $\delta_k : \mathbb{B}^n \times \mathbb{B}^t \rightarrow \mathbb{B}$.
In the following, we will assume that all FSMs have been derived from circuit designs. FSM states will be identified with assignments to a binary variable vector. Each entry in the vector corresponds to a latch in the sequential circuit. The same holds for inputs which can be regarded as a vector of binary signals. Every variable in the binary state vector (also called a *state bit*) will appear in two instances: a present state instance $x$ and a next state instance $y$. We write $\vec{x}$ to denote the current state vector and $\vec{y}$ to denote the next state vector which corresponds to the next state bits. The input vector is referred to as $\vec{\imath}$.

**Note:** *In later chapters some state variables will be explicitly named, e.g., namedvar. In such cases we write namedvar$'$ to denote the corresponding next state instance.*

### 4.3.1. Symbolic Representation

**Definition 4.9 (Characteristic function)**
*Let $(Q, I, O, \delta, \lambda, \vec{x}_0)$ be an FSM that has been derived from a sequential circuit with n latches. A set of states $A \subseteq Q$ can be described by its characteristic function*

$$\mathbf{A} : \mathbb{B}^n \to \mathbb{B}$$
$$\mathbf{A}(\vec{x}) = 1 \Leftrightarrow \vec{x} \in A$$

**Definition 4.10 (Transition relation)**
*Let $(Q, I, O, \delta, \lambda, \vec{x}_0)$ be an FSM that has been derived from a sequential circuit with n latches and t inputs. The transition relation of a latch k is given by the function*

$$\mathbf{T}_k : \mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B} \to \mathbb{B}$$
$$\mathbf{T}_k(\vec{x}, \vec{\imath}, y_k) = 1 \Leftrightarrow \delta_k(\vec{x}, \vec{\imath}) = y_k$$

*The transition relation of the FSM, i.e., the transition relation for all n latches, is given by the function*

$$\mathbf{T} : \mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B}^n \to \mathbb{B}$$
$$\mathbf{T}(\vec{x}, \vec{\imath}, \vec{y}) = 1 \Leftrightarrow \delta(\vec{x}, \vec{\imath}) = \vec{y}$$

The transition relation of an FSM with $n$ latches can be computed as the product of the transition relations of all latches, i.e.,

$$\mathbf{T}(\vec{x}, \vec{\imath}, \vec{y}) = \prod_{k=1}^{n} \mathbf{T}_k(\vec{x}, \vec{\imath}, y_k) \tag{4.3}$$

**T** is a characteristic function that represents the transition relation $T$. Since characteristic functions are switching functions, every set of FSM states, as well as the transition relation, can be represented as a BDD.
Figure 4.4 depicts a deterministic FSM for a sequential circuit with two latches and a single binary input signal. The equivalence of the two representations can be seen by investigating paths through the BDD representation. For example, the path $i_0 \longrightarrow y_1 \dashrightarrow y_0 \dashrightarrow 1$ expresses that all transitions which read $i_0 = 1$ lead to state $\vec{y} = 00$. Similarly, the transition from state $\vec{x} = 10$ to $\vec{y} = 10$ reading $i_0 = 0$ corresponds to the path $i_0 \dashrightarrow y_1 \longrightarrow y_0 \dashrightarrow 1$ .
The same idea for representing FSMs by the characteristic function of their transition relation can also be applied to non-deterministic FSMs, i.e., FSMs with a next-state function $\delta : Q \times I \to 2^Q$. However, symbolic state traversal methods for hardware verification usually consider deterministic FSMs because of the deterministic nature of sequential circuits. Therefore, we restrict this discussion to deterministic FSMs.

#### Simplified Definition of FSMs

We introduce two simplifications regarding the definition of FSMs. These simplifications will be used throughout the remainder of this work, in particular in Section 5.2 and Chapter 6.

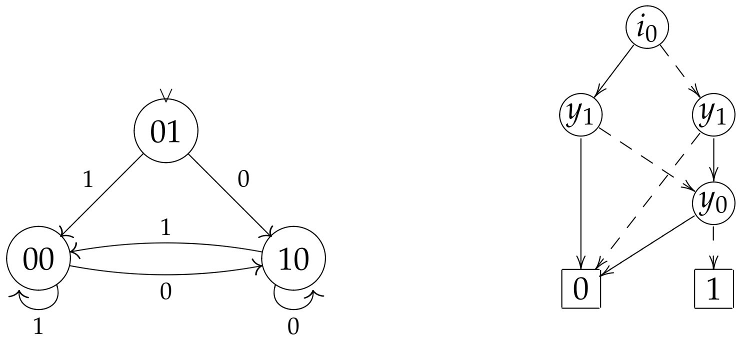


**Figure 4.4.:** Deterministic FSM and its BDD representation. The FSM corresponds to a sequential circuit with two latches $x_1, x_0$ and a single binary input signal $i_0$. The BDD is expressed over the input vector $\vec{\imath} = i_0$, the present state vector $\vec{x} = x_1, x_0$ and the next-state vector $\vec{y} = y_1, y_0$, i.e., $y_1$ and $y_0$ are the next-state instances of $x_1$ and $x_0$. Nodes for the present state variables $x_1, x_0$ have been removed by reduction. All transitions are clearly determined by input and next-state.

1. The output function of an FSM does not contribute to its transition relation. It will therefore be ignored for the purpose of state traversal.

2. The definition of a distinguished start state is often impractical for FSMs derived from sequential circuits. Computations can typically start from a number of different states. We will choose an appropriate set of start states depending on our application and context.

For the purpose of symbolic state traversal, the definition of an FSM will therefore be shortened to a 3-tuple $(Q, I, T)$ of a set of states $Q$ and inputs $I$ and a transition relation $T$ between the states of the FSM.

## 4.3.2. Image Computation

Image computation is at the core of symbolic state traversal. It denotes the computation of a set of successor states based on a given set of states and a transition relation. The set of successor states contains all states that are reachable in one transition step for any input. In order to give a formal definition of image computation we need to introduce the notion of existential quantification.

**Definition 4.11 (Existential quantification)**
*For $f \in \mathbb{B}_n$ the existential quantification with respect to the variable $x_i$ is defined by*

$$(\exists x_i)[f] = f_{x_i} + f_{\overline{x_i}}$$

The notion of quantification originates from the following equivalence where the symbol $\exists$ on the right side of the equivalence denotes the corresponding quantifier from predicate calculus:

$$(\exists_{x_i} f)[x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n] = 1 \Longleftrightarrow \exists_{x_i}[f(x_1, \ldots, x_n) = 1]$$

Note that $(\exists x_i)[f]$ denotes a switching function which no longer depends on the variable $x_i$. A generalized quantification operator over several variables is used in the following definition of image computation.

**Definition 4.12 (Image)**
*Let $(Q, I, T)$ be an FSM and $A \subseteq Q$ a set of FSM states. The image of $A$ under $T$, i.e., the set of states that can be reached from $A$ by a single transition according to $T$, is defined as:*

$$\begin{aligned}&\mathbf{Img} : (\mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B}^n \to \mathbb{B}) \times (\mathbb{B}^n \to \mathbb{B}) \to (\mathbb{B}^n \to \mathbb{B})\\ &\mathbf{Img}(\mathbf{T}, \mathbf{A})(\vec{y}) = (\exists \vec{x}, \vec{\imath})[\mathbf{T}(\vec{x}, \vec{\imath}, \vec{y}) \cdot \mathbf{A}(\vec{x})]\end{aligned}$$

A straightforward application of image computation is reachability analysis. Given a set of FSM states, it computes all states that are reachable according to the FSM's transition relation. Let **Init** be a Boolean function denoting a set of initial states. The set of reachable states can then be computed symbolically by the following fixed point computation.

$$\begin{aligned}\mathbf{A}_0(\vec{x}) &= \mathbf{Init}(\vec{x})\\ \mathbf{A}_{k+1}(\vec{y}) &= \mathbf{A}_k(\vec{y}) + \mathbf{Img}(\mathbf{T}, \mathbf{A}_k)(\vec{y})\end{aligned} \tag{4.4}$$

Image computation is the core operation of symbolic model checking algorithms and its efficient implementation has been the topic of many research papers [CBM90, BCL$^+$94, RAB$^+$95]. The crucial point is to avoid the blow-up of intermediate BDDs, which may result from computing the product of the transition relation with the current set of states. This blow-up may even happen in cases where the input and output of the image computation have compact BDD representations. A particularly efficient optimization is the partitioned representation of the transition relation. The idea is to represent the transition relation of Equation 4.3 as the product of partial relations. The partial relations are applied one after the other during image computation. This strategy allows to eliminate some variables by quantification before multiplication with the next partition and significantly reduces the intermediate memory consumption [RAB$^+$95].
The actual implementation details of image operators are quite complex. Fortunately, efficient implementations are freely available and can be used in an opaque way. Only the symbolic transition relation and the BDD representation of a set of FSM states have to be provided in a compatible (e.g., partitioned) format.

# CHAPTER 5

# Static Worst-Case Execution Time Analysis

Static *worst-case execution time* analysis (or WCET analysis for short) is a method for computing safe upper bounds on the execution times of single, uninterrupted tasks. It relies on abstract interpretation for computing invariants of the execution behavior from which execution time bounds can be deduced. In contrast to measurements, the method is exhaustive in terms of input data, program paths and hardware states, and therefore sound. Further, it is fully automatic but, like all static analyses that compute abstract invariants of undecidable concrete semantics, it is not complete. Hence, the method may fail to compute WCET bounds for certain programs without help from a user.

Safe and precise WCET analysis has to consider not only the program but also the hardware on which the program is executed. To this end, it requires a hardware model which expresses the complex interactions between the various performance enhancing hardware features such as caching and pipelining. Further, the analysis cannot consider a representation of the program in a high-level programming language. Instead, it must start from the level of fully linked binary executables where all required information (e.g., absolute hardware addresses, instruction sequence after compiler transformations) is available. The complexity of the analysis requires a careful design. Serious WCET tools are based on a combination of program transformations and dedicated analyzers that exchange information, e.g., semantic invariants. However, the growing complexity of the software and hardware in embedded systems tends to exacerbate the difficulties that are involved with WCET analysis. The employed hardware models are particularly affected by this evolution.

Static WCET analysis is used in the design and certification stages of industrial safety-critical real-time embedded systems, e.g., for electronic fly-by-wire systems [SLH+05, TSH+03]. Commercial implementations of the analysis are available [Abs00].

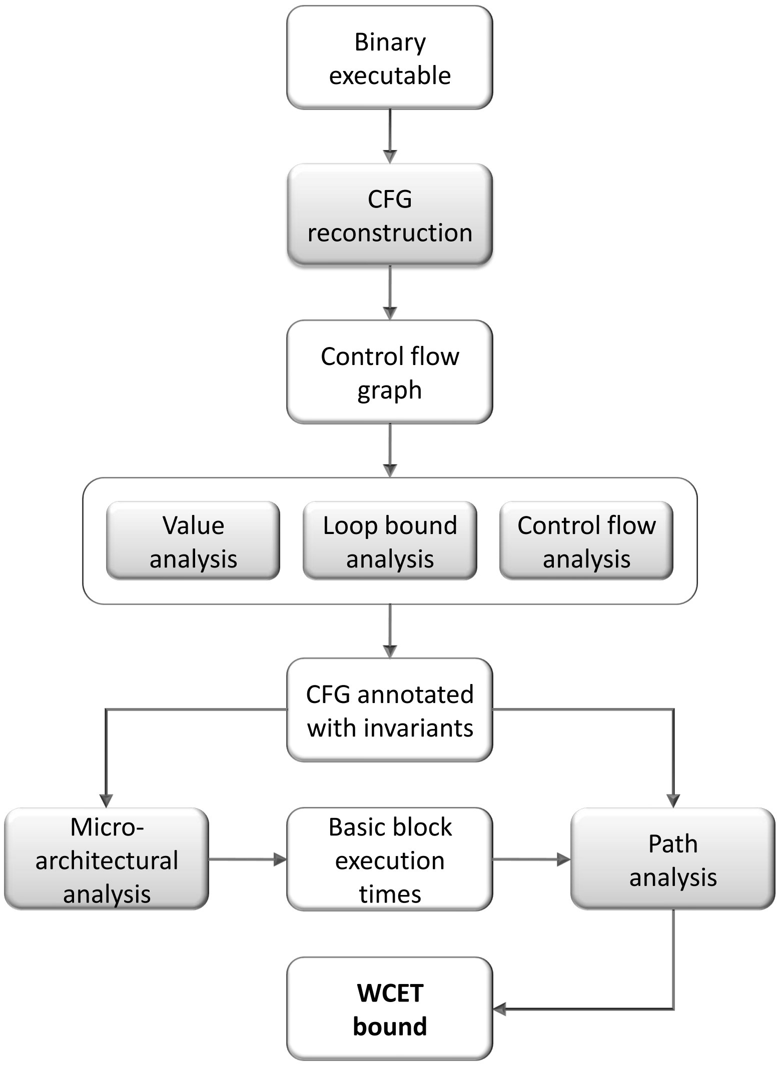


**Figure 5.1.:** General structure of static WCET analyzers.

## 5.1. A Framework for Static WCET Analysis

Figure 5.1 shows the general structure of static WCET analysis tools as described in [WEE+08]. The analysis structure is represented as a directed graph. Gray nodes correspond to different static analyses and program transformations. White nodes correspond to program representations and semantic invariants. Edges indicate the flow of information. Each analysis or transformation reads one program representation and produces another representation that may be enriched with invariants. The analysis starts from a binary executable and produces a WCET bound for the analyzed task. We present the involved analyses and transformations in top-down order with respect to Figure 5.1.

### 5.1.1. CFG Reconstruction

Safe and precise WCET analysis must consider the program execution with respect to a concrete hardware. This approach requires information that can only be obtained from fully linked binary executables. The CFG reconstruction phase uses a *binary decoder* that decompiles the text segment of the executable into a sequence of assembly instructions. It uses many heuristics and compiler-specific patterns to distinguish instructions from embedded data such as branch tables and constants. The reconstruction of the control flow can be performed in bottom-up order [The03]. Individual assembly instructions are grouped into basic blocks. Branch and call targets are extracted from the corresponding instructions and branch tables. Based on this information, the basic blocks are connected into basic block graphs (Definition 3.7, page 21) for procedures. The obtained procedures are then linked into a supergraph (Definition 3.11, page 25). The heuristics and patterns for extracting targets of computed[1] branches and calls may sometimes fail. In such cases, user annotations are required.

### 5.1.2. Value and Control Flow Analyses

Value analysis computes overapproximations of the possible contents of registers and stack cells[2] for each program point using an interval analysis on the extended supergraph of the analyzed task. The employed interval analysis [CC77] is a context-sensitive interprocedural data flow analysis that expresses operations on registers using interval arithmetics. For the purpose of WCET analysis, the computed intervals can be further used to determine the address ranges of indirect memory accesses as well as loop bounds and infeasible paths.

#### Loop Bound Analysis

A typical program spends most of its execution time in loops. Precise loop bound information is therefore indispensable for static WCET analysis. Loop bound analysis computes loop bounds by using pattern matching, dedicated data flow analyses, and value analysis results. The analysis is not complete and may fail to determine bounds for some loops. In such cases, user annotations are required.

#### Control Flow Analysis

Most control flow decisions depend on the values of variables. Control flow analysis exploits value analysis and loop bound information to detect *infeasible paths* in the program. A path is considered to be absolutely infeasible if the analysis can prove

[1]A computed branch or call jumps to an address that has been computed into a register.

[2]Compilers reserve memory cells within the function stack frame for spilling variables that cannot be held in registers. With knowledge of the compiler, stack accesses can often be disambiguated precisely thereby allowing to track individual variables through memory. This can also be done for global static variables. Tracking variables in the heap is much more difficult and often infeasible.

that it cannot be reached in any execution context. In addition, the analysis also determines paths that are only infeasible in some execution contexts. Infeasible paths can be excluded from subsequent analyses to reduce their complexity.

### 5.1.3. Microarchitectural Analysis

Modern microprocessors employ various techniques to reduce the average execution time. Examples of such techniques are pipelining, caching, out-of-order execution, branch prediction, and speculation. Microarchitectural analysis performs a data flow analysis based on abstract representations of hardware states. The abstract hardware representation allows to model the effects of features like pipelining and caching on the instruction execution. In particular, it also allows to consider the, sometimes counter-intuitive, interaction between the different features. The results of the analysis are context-sensitive upper bounds on the execution time of all basic blocks of the analyzed program. Microarchitectural analysis is implemented as the reduced product of two abstract domains called *cache domain* and *pipeline domain*.

**The cache domain** is based on an abstract representation of cache contents. It can be applied to pure instruction or data caches and to mixed instruction and data caches. The regular structure of caches allows for very efficient abstractions. In particular, the domain features an efficient join operator which allows to join two cache states into a single, less precise cache state. Thus, the abstract domain is not a power set of cache states but it is based on a partial order between cache states with different precision. The cache domain is described in detail in [Fer97].

**The pipeline domain** represents not only the state of the processor pipeline, but also of other timing relevant features, such as speculation, branch prediction, buses, and memory controllers. It is based on an *abstract pipeline model* and a state of this model is called an *abstract pipeline state*. The irregularity of the considered features hinders efficient join operations. Instead, the abstract domain is the power set of abstract pipeline states. The pipeline domain of a complex microprocessor is described in [The04].

In this setup, updates of both analysis domains are interdependent. The order of memory accesses into the cache depends on the pipeline state while the interleaving of instructions in the pipeline also depends on the latency of memory accesses. Therefore, both domains operate in parallel and exchange information. Technically, the pipeline domain *drives* the cache domain. Whenever an abstract pipeline state accesses cached memory, the pipeline domain triggers a cache update. The cache domain returns classification information (cache hit or cache miss) which is used by the pipeline domain to determine the latency of the memory access. Instructions that depend on the fetched information may not proceed until the information is available. Thus, the memory latency influences the possible interleavings of instructions in the abstract pipeline state.

### 5.1.4. Path Analysis

Path analysis uses *implicit path enumeration* to compute a global worst-case path through the analyzed program. To this end, it generates an integer linear program (ILP) that relates the execution time bounds and relative execution frequencies of neighboring basic blocks in the CFG. The execution time bounds are obtained from the results of microarchitectural analysis. Execution time frequencies are deduced from loop bounds and control flow analysis information. The objective function of the generated ILP maximizes the execution time. Thus, the solution to this ILP gives a WCET bound for the whole program and outputs a WCET path. An example of this technique on a high-level language control flow graph is given in Section 2.2.2. Note that the computed WCET path might not be taken in any concrete program execution. However, it is guaranteed that its worst-case execution time is an upper bound of all possible program executions. A detailed description of path analysis for binary-level WCET analysis is found in [The02].

### 5.1.5. Abstractions and the Loss of Precision

Like any semantics-based static analysis, static WCET analysis must lose information in order to gain computability. The presented toolchain loses information at the following steps:

**Control flow representation.** The context-sensitive interprocedural control flow graph (extended supergraph) may over-approximate the possible concrete execution paths of the analyzed program. For example, an indirect procedure call via a function pointer might not be disambiguated precisely. Further, several loop iterations can be combined into a single abstract iteration. This merging of loop iteration contexts leads to a loss of precision w.r.t. the execution paths inside the loop body. Static analyses operating on this program representation therefore consider spurious program paths that cannot be taken in any concrete execution.

**Value analysis.** Value analysis computes intervals rather than precise values. Precise values might either be unavailable, e.g., for input data, or the analysis chooses to join results from different execution histories.

**Loop bounds.** Loop bounds for WCET analysis are upper bounds. Overestimation of loop bound analysis results may lead to an overestimation of the global WCET bound by the global path analysis.

**Infeasible paths.** The set of infeasible paths determined by control flow analysis is an underapproximation, i.e. some infeasible paths might not be detected. Consequently, the remaining set of feasible program paths is an overapproximation. Subsequent analyses may therefore consider further spurious execution paths.

| cycles | FETCH | DECODE | EXECUTE | WRITEBACK |
|---|---|---|---|---|
| 1 | A | | | |
| 2 | B | A | | |
| 3 | C | B | A | |
| 4 | D | C | B | A |
| 5 | | D | C | B |
| 6 | | | D | C |
| 7 | | | | D |
| 8 | | | | |

**Figure 5.2.:** Pipelined execution of a linear sequence of 4 instructions A,B,C,D. Each instruction requires 4 cycles to go from fetch to writeback. Pipelining overlaps the execution of neighboring instructions, thereby reducing execution time from 17 to only 8 cycles[3].

**Cache abstraction.** Similar to value analysis, the cache domain may choose to join results from different execution histories. This join operation is associated with a loss of precision. As a result, a precise classification of cache accesses into hits and misses may become impossible.

**Path analysis.** Path analysis combines upper bounds on the execution times of basic blocks without considering the relationship between execution paths in neighboring basic blocks. Thus, it considers combinations that cannot occur in any concrete execution. However, the WCET bound of the chosen combination is always an upper bound of all possible concrete executions.

## 5.2. Pipeline Domains

Pipelining reduces the execution time of a program by overlapping the execution of individual instructions. Figure 5.2 shows the general concept using a simple pipeline with 4 stages: fetch, decode, execute, and writeback. Similar pipelines are implemented in many embedded processors. More complex processors often implement several such pipelines that operate in parallel. Complex pipelines may also implement more stages. The implementation of a pipeline is part of the microprocessor core design which is a sequential circuit. It has a state that is stored in latches, and an update logic that is given by the interconnection of logic gates. Updates of the pipeline state depend on the current state and on inputs. Inputs

[3]Execution time is counted until the last instruction has left the pipeline. Instruction D is still in the pipeline in cycle 7 and has left the pipeline in cycle 8.

| cycles | FETCH | DECODE | EXECUTE | WRITEBACK |
|---|---|---|---|---|
| 1 | A | | | |
| 2 | B | A | | |
| 3 | C | B | A | |
| 4 | D | C | B | A |
| 5 | | D | C | B |
| 6 | | D | C | |
| 7 | | | D | C |
| 8 | | | | D |
| 9 | | | | |

**Figure 5.3.:** In this example, the execution of instruction C depends on an operand of instruction B. The operand is only available when B reaches the writeback stage. C must wait for one cycle in the execute stage until the operand is available. Execution of A,B,C,D takes 9 cycles.

either deliver information about the executed program (instruction fetches) or about the state of other hardware components, e.g., caches or memory controllers. The evolution of the pipeline during program execution can be expressed by a sequence of pipeline states. Depending on the inputs and on the current state of the pipeline, instructions may either flow smoothly through the pipeline (see Figure 5.2), or the execution of an instruction may *stall* at a certain pipeline stage. Stall events are also known as *pipeline hazards*. Figure 5.3 shows an example where an instruction cannot proceed due to an unresolved data dependence. The execution is delayed by 1 cycle until the dependence is resolved. In general, pipeline hazards lead to an increase in execution time. WCET analysis must predict possible pipeline hazards in order to compute safe upper bounds on the execution time. This prediction is based on an overapproximation of the concrete pipeline semantics in terms of reachable pipeline states. The relationship between the concrete semantics and the predictions of pipeline analysis can be explained using FSMs as computational models. The concrete semantics is based on a *concrete pipeline* which is a deterministic FSM, whereas the abstract semantics is based on an *abstract pipeline* that is a non-deterministic FSM. In the definition of the concrete and abstract semantics we deliberately avoid the term *abstract pipeline model*. We reserve this term for actual implementations of the computational model.

### 5.2.1. Concrete Semantics

To define the semantics of the pipelined execution of a program, we introduce a general machine-level program representation.

**Definition 5.1 (Program)**
*A machine-level program $L$ is a totally ordered set of attributed machine instructions $l_0, l_1, \ldots, l_m$. The unique start instruction of $L$ is $l_0$. The set of terminal instructions (program exits) is given by $L_\chi \subset L$. The value of an attribute attrib at instruction $l$ is denoted by $l.attrib$. The total order is established by the numerical value of address attributes:*

$$l_0.addr < l_1.addr < \ldots < l_m.addr$$

*The set of all machine-level programs is denoted by $\mathcal{L}$.*

In a machine-level basic block graph, each basic block is also a program because the contained instructions appear in strictly increasing order of their addresses as required by Definition 5.1. The unique start instruction of a basic block is the instruction with the smallest address. The whole graph represents a program if the contained instructions are combined in increasing order of their address attributes. The pipelined execution of a program can be represented by a sequence of pipeline states that is called a *trace*. The following definition does not impose any particular constraint on traces. It allows for traces that do not correspond to any concrete execution because some sequences of pipeline states cannot occur. This slackness will be rectified after the concrete execution has been defined.

**Definition 5.2 (Trace)**
*Let $Q$ be the set of pipeline states and $q \in Q$ a single pipeline state. A trace $t$ is a finite sequence of pipeline states $t = q_0, q_1, \ldots, q_n$. The set of all possible traces is denoted by $\mathcal{T}$.*

The sequential circuit of the pipeline can be represented by an FSM (see Section 4.3). We do not define a distinct start state since a program may start from any feasible start state (see Section 4.3.1).

**Definition 5.3 (Concrete pipeline)**
*A concrete pipeline is a deterministic FSM $(Q, I, O, \delta, \lambda)$ with state space $Q$, input space $I$, output space $O$, a next-state function $\delta : Q \times I \to Q$, and an output function $\lambda : Q \times I \to O$. It can be given either by the hardware implementation or by a software model that accurately and completely represents all states and transitions of the actual hardware.*

During the execution of a program, a concrete pipeline interacts with many other hardware components. For example, if the execution of a certain instruction requires reading the contents of a specific memory address, a request is sent to the memory controller which returns the contents of that address, thereby possibly updating the state of associated caches. This exchange of information uses the input and output signals of the pipeline. In the following, we regard all other components as being part of a single black box that we call the *environment*. Like the pipeline, the environment can also be regarded as an FSM. We require this FSM to be deterministic which expresses the constraint that we only consider uninterrupted program executions for static WCET analysis. For a feasible start state, the trace of a pipelined program execution can be computed by repeated application of the next-state functions of both interacting FSMs.

**Definition 5.4 (Environment)**
*Let $(Q, I, O, \delta, \lambda)$ be a concrete pipeline. An environment is a deterministic FSM $(\Sigma, O, I, \varphi, \rho)$ that is connected with a pipeline FSM via its inputs and outputs. Updates of the environment state are represented by a function $\varphi : \Sigma \times O \to \Sigma$ and its output function $\rho : \Sigma \times O \to I$ generates inputs for the pipeline FSM.*

For a given concrete pipeline, environment, and program, traces can be computed by the interleaved execution of the next-state functions of the two FSMs. Let $\epsilon$ be the empty input and $q_s$ be the initial state of the trace. The initial environment state is obtained by a function $\xi : \mathcal{L} \times \Sigma \to \Sigma$ which loads the program $L$ into the empty environment state $\sigma_s \in \Sigma$. The trace states $q_0, q_1, \dots, q_n$ can then be computed by the following recursive algorithm:

$$\begin{array}{llll} q_0 = q_s & \sigma_0 = \xi(L, \sigma_s) & o_0 = \lambda(q_0, \epsilon) & i_0 = \rho(\sigma_0, o_0) \\ q_k = \delta(q_{k-1}, i_{k-1}) & \sigma_k = \varphi(\sigma_{k-1}, o_{k-1}) & o_k = \lambda(q_{k-1}, i_{k-1}) & i_k = \rho(\sigma_{k-1}, o_{k-1}) \end{array} \tag{5.1}$$

The following definition associates the trace computation of Equation 5.1 with an execution function which takes a program and a start state. The execution function is specific to a certain pipeline FSM and environment FSM. Since the following text only considers one pipeline FSM and environment FSM at a time, there is no need to distinguish between execution functions for different pipelines and environments.

**Definition 5.5 (Execution)**
*Let $\mathcal{L}$ be the set of programs and $Q$ the set of states of a concrete pipeline. An execution of a given program by the concrete pipeline model with state space $Q$ is expressed by the function*

$$exec : \mathcal{L} \times Q \to \mathcal{T}$$

*The function implements the algorithm of Equation 5.1 using a concrete pipeline with state space $Q$ and a compatible environment.*

Program execution cannot start from arbitrary pipeline states. We require that the first state of a trace for a program $L$ fetches the first instruction of $L$. The execution of a program ends if a terminal instruction has just left the pipeline. We say that the instruction is *retired*.

**Definition 5.6 (Valid trace, feasible start state)**
*Let $(Q, I, O, \delta, \lambda)$ be a pipeline and $L = l_0, l_1, \dots, l_m$ be a program. For a start state $q_0 \in Q$ the trace $q_0, q_1, \dots, q_n = exec(L, q_0)$ is valid for $L$ if and only if*

- *the first instruction $l_0$ is fetched in state $q_0$, and*
- *the last instruction $l_k$ with $k \leq m$ is retired in state $q_n$ and $l_k$ is a terminal instruction, i.e., $l_k \in L_\chi$.*

*If $q_0, q_1, \dots, q_n$ is a valid trace for $L$, then $q_0$ is called a feasible start state for $L$.*

| | FET | DEC | EXE | WBK |
|---|---|---|---|---|
| | A | | | |
| | B | A | | |
| | C | B | A | |
| | D | C | B | A |
| $\llbracket n_A \rrbracket \;:=$ | | D | C | B |
| $\llbracket n_B \rrbracket \;:=$ | | | D | C |
| $\llbracket n_C \rrbracket \;:=$ | | | | D |
| $\llbracket n_D \rrbracket \;:=$ | | | | |

**Figure 5.4.:** Assignment of pipeline states to CFG nodes that correspond to retired instructions.

**Note:** *We only consider valid traces of correct programs. Hence, for the remainder of this text we prefer the shorter term trace whenever we mean valid trace.*

The concrete semantics of the pipelined execution of a program can be expressed by assigning reachable pipeline states to the nodes of the associated CFG. We associate each program point with the pipeline state where the corresponding machine instruction is retired. Hence, the final state of a trace is associated with the last instruction of the executed program. The considered semantics is equivalent to the path semantics (Definition 3.8) obtained by chaining a transfer function that computes partial traces at the level of single instructions. Figure 5.4 shows an example of the association of trace states to program points.
To express the *execution time* of a program that is subject to pipelined execution, we enrich the concrete semantics domain by a cycle counter. The counter is incremented whenever the pipeline FSM has finished its transitions for one clock cycle. The counter can be added to the pipeline. At the end of program execution, i.e., when a terminal instruction is retired, the value of the cycle counter gives the program execution time. A pipeline can be designed in a way that the number of transitions per clock cycle is a constant $n$. This can be achieved by adding $\epsilon$-transitions wherever the number of transitions depending on one clock tick is less than $n$. Hence, the final value of the cycle counter is equivalent to the number of states in the corresponding trace of the concrete pipeline divided by $n$. In the following, we assume that $n = 1$. The application to cases where $n > 1$ is trivial.

## Basic Block Traces

Traces can also be computed at the level of individual basic blocks (remember that a basic block is also a program according to Definition 5.1). Due to the overlapping of instructions in the pipeline, the final state of a basic block trace is typically not a feasible start state for the subsequent basic block. Its execution must start from

| Block | State | FET | DEC | EXE | WBK |
|---|---|---|---|---|---|
| B1={A,B} | 1 | A | | | |
| | 2 | B | A | | |
| | 3 | C | B | A | |
| | 4 | D | C | B | A |
| | 5 | | D | C | B |
| | 6 | | | D | C |
| B2={C,D} | 1 | | | | D |
| | 2 | | | | |

**Figure 5.5.:** Applying *exec* to two basic blocks B1 and B2 that are connected by a fall-through edge. B1 contains the first two instructions A and B while B2 contains the subsequent instructions C and D. The depicted trace for B2 is partial. The complete trace for B2 would start from state 3 of B1.

an earlier state where its first instruction is fetched. The resulting overlapping of basic block traces corresponds to the overlapping of instructions in the pipeline. The overlapping of basic block traces can be avoided if one allows partial traces where the first state need not be a feasible start state for a block $b \in B$. It suffices to require that it is a *valid* state for $b$, i.e., that it appears in a valid trace of $b$. An example is depicted in Figure 5.5.

**Definition 5.7 (Basic block execution)**
*Let $G = (B, E, n, s)$ be a basic block graph and $Q$ a set of pipeline states. The execution of a basic block $b \in B$ starting from a valid pipeline state is expressed by a function that computes a trace $t \in \mathcal{T}$:*

$$exec : B \times Q \rightarrow \mathcal{T}$$

*The length of a basic block trace $t$ is denoted by $|t|$. The number of execution cycles of a block $b$ in the execution context defined by the start state equals $|t|$.*

## 5.2.2. Abstraction

A complete and precise representation of a modern pipeline is very complex. An efficient domain for pipeline analysis must be based on an abstract representation that simplifies the pipeline without underestimating the execution time contribution of pipeline accidents. Further, it should also capture the benefits of pipelining in order to provide tight WCET bounds. Thesing [The04] constructed an efficient abstract pipeline analysis for the MPC 755 processor [Mot97, Fre01] and proved its correctness with respect to the concrete execution model. We show the general concept of pipeline abstraction without going into the details of a specific processor.

**Note:** *From this point on we use the simplified FSM definition as introduced in Section 4.3.1.*

**Definition 5.8 (Abstract pipeline)**
*Let the FSM $(Q, I, \delta)$ be a concrete pipeline with a deterministic next state function $\delta : Q \times I \to Q$. The FSM $(\widehat{Q}, \widehat{I}, \widehat{\delta})$ is called an abstract pipeline for $(Q, I, \delta)$ if its next-state function is non-deterministic, i.e., $\widehat{\delta} : \widehat{Q} \times \widehat{I} \to 2^{\widehat{Q}}$, and if states of $Q$ are mapped to states of $\widehat{Q}$ by a total function $\alpha_0 : Q \to \widehat{Q}$. It is a correct abstraction if it holds that*

$$\alpha_0(q) = \widehat{q} \Rightarrow \alpha_0(\delta(q, i)) \in \widehat{\delta}(\widehat{q}, \widehat{\imath})$$

*whenever $\widehat{\imath}$ is an abstract input that corresponds to $i$.*

In general, the abstract pipeline has fewer states than the concrete pipeline and many concrete states can be mapped to a single abstract state. Consequently, the set of non-deterministic transitions of the abstract pipeline has also fewer elements than the set of deterministic transitions of the concrete pipeline. For example, abstract pipelines discard the representation of register files and arithmetic units since the corresponding logic is efficiently handled by the value analysis. All arithmetic transitions are mapped to very few non-deterministic transitions in the abstract pipeline. The non-determinism handles the effects of imprecise inputs and the abstraction of hardware components. The correspondence between abstract and concrete inputs in Definition 5.8 also depends on properties of the value and control flow analyses.

**Definition 5.9 (Pipeline domain, Pipeline abstraction)**
*Let $(Q, I, \delta)$ be a concrete pipeline and $(\widehat{Q}, \widehat{I}, \widehat{\delta})$ be a corresponding abstract pipeline. Further, let $\mathcal{D}^{\flat}$ and $\mathcal{D}^{\sharp}$ be two complete lattices defined as $\mathcal{D}^{\flat} \equiv (2^{Q}, \subseteq, \cup, \cap, \emptyset, Q)$ and $\mathcal{D}^{\sharp} \equiv (2^{\widehat{Q}}, \subseteq, \cup, \cap, \emptyset, \widehat{Q})$, respectively. $\mathcal{D}^{\flat}$ and $\mathcal{D}^{\sharp}$ are called pipeline domains. A pipeline abstraction is a pair of functions $(\alpha, \gamma)$ where $\alpha$ is an abstraction function defined as*

$$\begin{aligned} &\alpha : \mathcal{D}^{\flat} \to \mathcal{D}^{\sharp} \\ &\alpha(X^{\flat}) = \{\alpha_0(q) \mid q \in X^{\flat}\} \end{aligned}$$

*and $\gamma$ is the corresponding concretization function defined as*

$$\begin{aligned} &\gamma : \mathcal{D}^{\sharp} \to \mathcal{D}^{\flat} \\ &\gamma(X^{\sharp}) = \{q \mid \alpha_0(q) \in X^{\sharp}\} \end{aligned}$$

$\mathcal{D}^{\flat}$ and $\mathcal{D}^{\sharp}$ are both power set domains and therefore complete lattices by construction as discussed in Section 3.1. We show that $(\alpha, \gamma)$ is a Galois connection between both domains.

**Theorem 5.1 (Galois connection between pipeline domains)**
*Let $\mathcal{D}^b$ and $\mathcal{D}^\sharp$ be a concrete and an abstract pipeline domain, respectively. A pipeline abstraction $(\alpha, \gamma)$ is a Galois connection between $\mathcal{D}^b$ and $\mathcal{D}^\sharp$.*

**Proof** The function pair $(\alpha, \gamma)$ is a Galois connection if it ensures local consistency, i.e., if for all $X^b$ , $X^\sharp$ it holds that:

$$\alpha(X^b) \sqsubseteq^\sharp X^\sharp \iff X^b \sqsubseteq^b \gamma(X^\sharp)$$

By using the definitions of $\alpha$ and $\gamma$ and the fact that both domains are power sets, i.e., the partial order operator is the subset operator, this equivalence can be rewritten as

$$\{\alpha_0(q) \mid q \in X^b\} \subseteq X^\sharp \iff X^b \subseteq \{q \mid \alpha_0(q) \in X^\sharp\}$$

The truth of this equation can be easily seen by set-theoretic reasoning. ■

## 5.2.3. Abstract Semantics

The semantics of pipelined program execution can be safely approximated by a data flow analysis on the program's CFG. For each basic block, the transfer function computes a set of leaving pipeline states from a set of incoming pipeline states. This computation involves the construction of abstract traces that represent the pipelined execution of the current basic block. In contrast to concrete traces (see Definition 5.2), traces in the abstract domain are lifted to *sets* of abstract pipeline states.

**Definition 5.10 (Abstract pipeline trace)**
*Let $\mathcal{D}^\sharp$ be the domain of an abstract pipeline. Further let L be a program. $A_k^\sharp \in \mathcal{D}^\sharp$ denotes a subset of abstract pipeline states. An abstract pipeline trace $\widehat{t}$ is a sequence of sets of abstract states $\widehat{t} = A_0^\sharp, A_1^\sharp, \dots, A_n^\sharp$. The set of all possible abstract traces is denoted by $\widehat{\mathcal{T}}$.*

Abstract pipeline traces do not contain information about the predecessor-successor relationship between abstract states in two successive sets of the trace. This definition corresponds closely to the symbolic-state pipeline analysis domain that is presented in Chapter 6. In contrast the explicit-state pipeline analysis presented in [The04] computes an abstract trace as a set of traces rather than a trace of sets. This difference between the two approaches is visualized by Figure 5.6 and Figure 6.5.
We define the abstract execution function of basic blocks without an explicit argument for handling inputs. Instead, all inputs are obtained from instruction attributes that store the results of preceding analyses (like value analysis, for example); the abstract execution function implicitly retrieves its inputs by querying the instructions of the basic block. We also make the simplifying assumption that no cache analysis (see Section 5.1.3) is present. We address the extension to architectures with caches in Section 6.5.

**Definition 5.11 (Abstract basic block execution)**
*Let $G = (B, E, n, s)$ be a basic block graph and $\mathcal{D}^b$, $\mathcal{D}^\sharp$ be a concrete and an abstract pipeline domain that are connected by a pipeline abstraction $(\alpha, \gamma)$. An abstract state $A^\sharp \in \mathcal{D}^\sharp$ is called a valid start state for a basic block $b \in B$ if it holds that*

$$\forall q \in \gamma(A_0^\sharp) : q \textit{ is a feasible start state for } b.$$

*The abstract execution of a basic block is expressed by a function*

$$\widehat{exec} : B \times \mathcal{D}^\sharp \to \widehat{\mathcal{T}}$$

*such that for each basic block $b \in B$ and each valid start state $A_0^\sharp \in \mathcal{D}^\sharp$ the final state of the resulting abstract trace $A_0^\sharp, A_1^\sharp, \ldots, A_n^\sharp$ satisfies the following condition:*

$$\forall q \in \gamma(A_n^\sharp) : \textit{ the instruction retired in } q \textit{ is the terminal instruction}^4 \textit{ of } b.$$

Pipeline analysis is implemented as a data flow analysis that computes reachable sets of pipeline states for all program points. According to Section 3.2, data flow analysis requires a data flow domain, a transfer function, an upper bound operator and an equality test. For pipeline analysis we use the following implementations:

1. The data flow domain is the domain of abstract pipeline states $\mathcal{D}^\sharp \equiv (2^{\widehat{Q}}, \subseteq, \cup, \cap, \emptyset, \widehat{Q})$.

2. The implementation of the transfer function $tf^\sharp : E \to \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is based on the abstract execution of basic blocks. It takes an incoming set of abstract states and an edge $(s, t) \in E$ and constructs the abstract trace of the source block $s$ using the abstract execution function $\widehat{exec} : B \times \mathcal{D}^\sharp \to \widehat{\mathcal{T}}$. Abstract states that retire the terminal instruction of $s$ are propagated to $t$ if they have been computed by a sequence of transitions that leads to $t$ according to the semantics of the abstract pipeline.

3. The upper bound operator $\sqcup^\sharp$ is implemented by set union, i.e. $X^\sharp \sqcup^\sharp Y^\sharp \equiv \widehat{X} \cup \widehat{Y}$ for $\widehat{X}, \widehat{Y} \subseteq \widehat{Q}$.

4. The equality test $=^\sharp$ for elements of $\mathcal{D}^\sharp$ checks whether both sets contain exactly the same abstract states: $X^\sharp =^\sharp Y^\sharp$ *iff* $\forall \widehat{x} \in \widehat{X} : \widehat{x} \in \widehat{Y}$ *and* $|\widehat{X}| = |\widehat{Y}|$.

## 5.2.4. WCET Bounds for Basic Blocks

The computation of reachable pipeline states is not the primary purpose of pipeline analysis. For WCET analysis we are rather interested in WCET bounds for basic blocks. Indeed, WCET bounds can be derived from the lengths of the computed abstract traces. Definition 5.7 states that the execution time of a basic block is given by the length of its trace. The last iteration of the data flow fixed point algorithm

[4] A basic block is a program that has only a single terminal instruction.

computes abstract basic block traces with sets of incoming pipeline states that safely approximate the reachable states of any concrete execution. The length of such a trace is a safe upper bound for the WCET of the associated basic block. The underlying assumption is that the abstract pipeline, as well as all input information (the results of control flow and value analysis), is conservative with respect to the execution time.

**Definition 5.12 (Conservative)**
*Let $(Q, I, \delta)$ be a concrete pipeline and $(\widehat{Q}, \widehat{I}, \widehat{\delta})$ the corresponding abstract pipeline. The set $B^*$ denotes the set of all basic blocks of all programs. The abstract pipeline is conservative with respect to the execution time if for all basic blocks $b \in B^*$ and states $q \in Q$ that appear in a valid trace of $b$ the following condition holds:*

$$|exec(b, q)| \leq |\widehat{exec}(b, \widehat{X})|, \forall \widehat{X} : \alpha_0(q) \in \widehat{X} \tag{5.2}$$

The execution time bounds of basic blocks can be safely derived from the abstract semantics of pipeline analysis. The computed bounds overestimate the execution times of any corresponding concrete execution if the abstract pipeline is indeed conservative.

## 5.3. The State Explosion Problem

The abstract semantics of pipeline analysis can be computed by different methods as already mentioned in Section 5.2.3. This section gives an informal description of the explicit-state approach that is taken by state-of-the-art implementations. The explicit-state approach is susceptible to the *state explosion problem in WCET analysis*. We describe this problem and motivate the use of symbolic methods for pipeline analysis.

### 5.3.1. State Explosion in WCET Analysis

Static analyses lose information in order to gain computability. We have pointed out in Section 5.1.5 how information is lost in various stages of WCET analysis. The loss of information leads to a loss of precision, i.e., to a coarser WCET bound. This is acceptable if the resulting bound is still *tight enough*. However, the loss of precision also has an effect on the complexity of pipeline analysis. We give an example to illustrate this claim by considering the interaction between value analysis and pipeline analysis.
Value analysis (see Section 5.1.2) computes the possible contents of registers at each program point. Whenever the hardware accesses memory using a register value as an address, value analysis results provide information about the possible addresses that are accessed at run-time. In embedded systems, different memory regions often significantly differ in their access latencies. Memory accesses may either hit scratch-pad memory, flash modules, cached memory areas, or memory-mapped external devices. Some accesses, like accesses into a cached memory area, may also

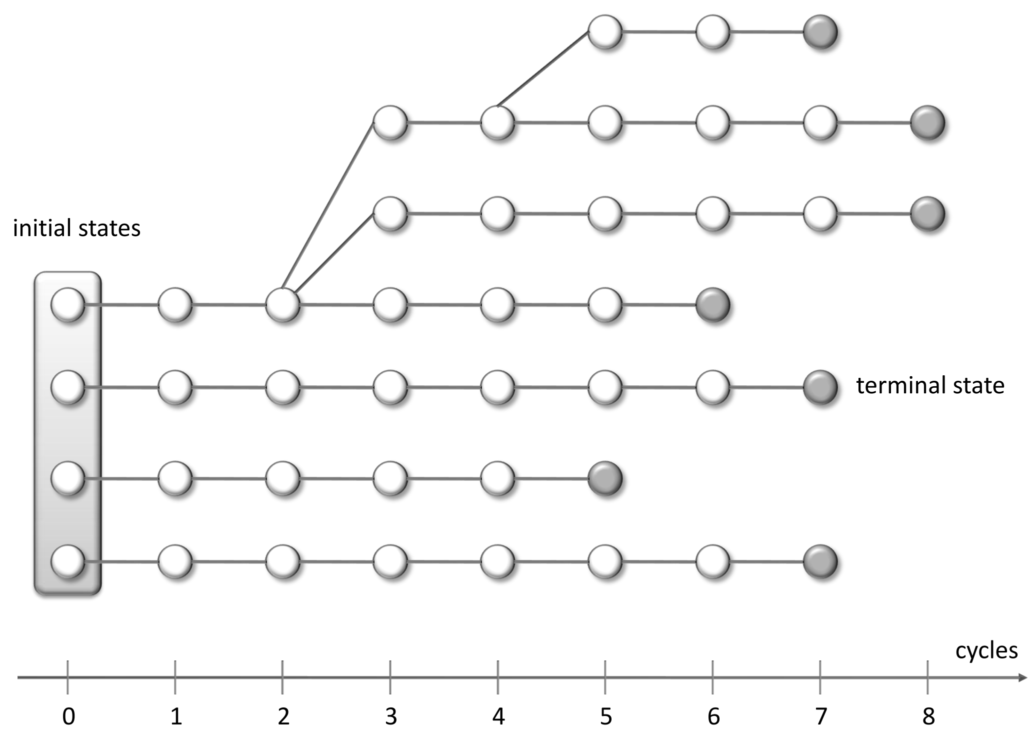


**Figure 5.6.:** Explicit-state computation of an abstract pipeline trace for a single basic block in DFS order. Non-deterministic transitions are due to imprecise value analysis, cache, or control flow information. The abstract trace is computed as a set of traces rather than as a trace of sets, i.e., the analysis computes only one path at a time.

alter the hardware state. For modern architectures where all memory accesses are sent over complex buses, all memory accesses affect the bus state and therefore the hardware state. Hence, whenever the state traversal for computing abstract pipeline traces encounters a memory access that cannot be disambiguated precisely, it has to assume several possible successor states. The FSM that represents the abstract pipeline needs to be non-deterministic.

State-of-the-art implementations of pipeline analysis represent the abstract pipeline by its transition function written in a high-level programming language. The transition function operates on an explicit representation of pipeline states. Abstract traces are computed in DFS order as depicted in Figure 5.6. Memory consumption and computation time of the analysis grow linearly with the number of reachable states due to the explicit state representation. In certain cases, the analysis can become infeasible in practice [The04]. This problem has been termed the state explosion problem in WCET analysis.

## 5.3.2. Timing Anomalies and Domino Effects

The computation of basic block traces is a state space exploration because of the non-deterministic nature of the abstract pipeline. The reachable state space is constrained by the precision of the available input information from the value, cache, and control flow analyses. For the approximation of safe WCET bounds, this state space exploration must be exhaustive. It is not safe to discard states by making local assumptions about the worst case, nor to account for an ignored trace by adding a constant to the WCET bound. The reason for this is that modern, complex processor hardware exhibits counter-intuitive execution time behavior. Such behavior has also been termed *timing anomaly* [LS99a].

**Definition 5.13 (Timing anomaly)**
*A timing anomaly is a situation where a local decrease (increase) of latency leads to a global increase (decrease) in execution time.*

A well-known example of a timing anomaly affects the Motorola PowerPC 755 [Mot97, Fre01]. The situation is depicted in Figure 5.7. A cache hit (the situation shown in the upper half of the figure) in the execution of instruction *A* has a lower latency compared to a cache miss (the situation shown in the lower half of the figure). In the second case, instruction *B* has to wait for *A* to complete due to a data dependence. Since the integer unit (IU) is not occupied, the superscalar processor immediately starts the execution of the independent instruction *C* (out-of-order execution). On a larger scale, the cache miss reduces the execution time since instructions *D* and *E* that depend on *C* can be executed earlier.
The notion of a *domino effect* denotes a situation where:

1. There is a hardware event that causes a timing accident (e.g., a cache miss) during the execution of a certain code sequence.

2. A trace that starts with this event is longer than other traces of the same code sequence. This trace is called the *critical* trace.

3. The critical trace leads to the same event that caused the initial timing accident.

If the corresponding code sequence is executed in a loop, the execution time difference between an execution that runs into the critical trace and other executions cannot be bounded by a constant. It depends on the number of loop iterations. For this reason, domino effects have also been called *unbounded timing anomalies*. Several examples of domino effects have been observed in real hardware, e.g., for the pipeline of the MPC 755 [Sch03] and for the PLRU cache replacement strategy [Ber06].

## 5.3.3. The Need for More Efficient Pipeline Domains

State-of-the-art implementations of pipeline analysis do not explicitly construct the FSM of the abstract pipeline. It is specified only in terms of its transition function

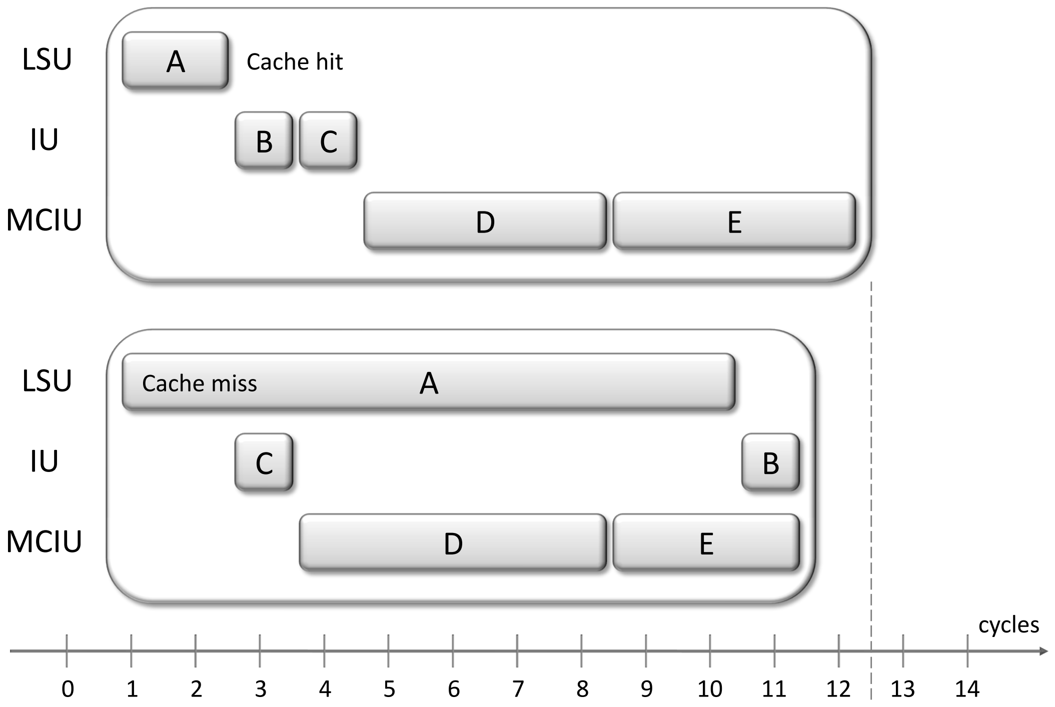


**Figure 5.7.:** Illustration of a timing anomaly for the Motorola PowerPC 755. There is a data dependence between instructions A and B, but not between instructions B and C. In the second case, the superscalar processor therefore executes C before B while waiting on A (which is delayed by a cache miss). The cache miss reduces the overall execution time because the sequence C,D,E starts one cycle earlier, thanks to the reordering of B and C.

written in a high-level programming language (usually `C` or `C++`). This design avoids the state explosion problem involved with the construction of the FSM and allows to specify large, complex pipelines. The drawback of this construction is the fact that the transition function must be called explicitly for every reachable pipeline state. Hence, pipeline states must also be represented explicitly. The preceding sections have shown that the computation of abstract pipeline traces requires an exhaustive traversal of the reachable state space of the abstract pipeline. The size of the reachable state space depends on the precision of information from other analyses in the WCET analysis framework. The achievable precision of static analyses is limited because of the undecidability of all non-trivial questions about the concrete program semantics. It is further limited because of uncertainty about inputs. Input information in embedded systems usually depends on the state of external physical systems. For example, an electronic flight control system must consider the current

altitude and speed of the aircraft. Such information cannot be known precisely at analysis time.
We have explained in Section 5.3.1 how imprecise knowledge about the exact address of a memory access increases the reachable state space. The following numbers give an idea of the size of the problem in practice. They have been obtained by close examination of available commercial pipeline analyzers [Abs00].

**The Infineon TriCore 1** is an embedded processor of medium complexity. A single unclassified memory access leads to at most 64 successor states per current state in its abstract pipeline analysis.

**The Freescale MPC 7448** is the most complex embedded processor for which a commercial WCET analysis based on abstract interpretation is available today. In its abstract pipeline analysis, a single unclassified memory access leads to up to 1000 successor states per current state.

Another point to consider is that pipeline analysis must already start from a *set* of feasible pipeline states. It is not safe to assume that the empty pipeline is the worst-case for execution time [The04]. Finally, the trend towards more complex processor architectures in safety-critical embedded systems leads to abstract pipelines with substantially larger reachable state spaces.
To conclude this chapter, let us reason about the chances of coming up with a pipeline domain that is not based on a power set construction. Such a domain could be expected to be more efficient since it might join two pipeline states into a single more abstract state. However, the safe approximation of WCET bounds requires that elements in the underlying lattice are ordered with respect to the *global* WCET bound, i.e., replacing an element by a less precise element must not decrease the global execution time bound. The notion of timing anomalies shows that it is inherently difficult to predict the global timing effect of a local decision. Any pipeline domain that relies on local decisions must prove that it makes no assumption that decreases the global WCET bound. This is equivalent to proving the absence of timing anomalies in the considered program and execution contexts. So far, research on timing anomalies only succeeded in automatically identifying individual cases of anomalies [EPB+06]. Therefore, pipeline domains that do not rely on power sets of abstract states are currently not in sight.

CHAPTER 6

# Symbolic Representation of Pipeline Domains

In Chapter 5 we discussed how pipeline domains for WCET analysis approximate the collecting semantics of program execution by a state traversal of complex microprocessor models. We showed that the existence of timing anomalies and domino effects forbids to make simplifying assumptions that would reduce the reachable state space. Since state-of-the-art implementations of pipeline domains traverse the reachable state space in DFS order using an explicit representation of pipeline states, memory consumption and computation time of the analysis grow linearly with the number of reachable states. If the reachable state space grows too large, e.g., because of the limited precision of knowledge about possible register contents, the analysis can become infeasible in practice. We address the state explosion problem in static WCET analysis by storing and manipulating pipeline states using BDDs. The approach is inspired by symbolic model checking which has been successfully applied to the analysis of hardware and software. But in contrast to existing applications of model checking, WCET analysis considers *both* the software and the processor simultaneously. The modular WCET analysis architecture of Chapter 5 alleviates the arising complexity by careful modeling and abstraction. However, at the level of pipeline analysis, the interplay of hardware and software – as well as the interaction with the different analyzers – remains complex, conceptually and in terms of implementation. This chapter discusses the design of a symbolic representation of pipeline domains. We want to arrive at an implementation that completely avoids the explicit enumeration of pipeline states and thereby alleviates the state explosion problem in WCET analysis. At the same time, the following constraints have to be respected:

1. The pipeline analysis must cooperate with the other analyses of the WCET analysis framework. Information must be imported from and exported to the extended supergraph.

2. The approach must scale to serious pipeline models and programs.

Due to these constraints, the analysis cannot be modeled in an off-the-shelf model checking tool. The necessary exchange of analysis information with the WCET framework requires not only a translation of analysis information into a symbolic representation. The state exploration process itself also needs to be guided by the structure of the supergraph. We will see that this exploration strategy – which proceeds basic block-wise – also provides opportunities for optimizations that allow the analysis to scale to very large programs. Finally, interfacing an abstract cache analysis is particularly difficult and requires a tight integration between the symbolic representation and the interface of the cache analysis that cannot be implemented with existing tools.

## 6.1. Considered Hardware Features

The approach that is presented in this chapter does not put special restrictions on the analyzed software. It accepts any industrial real-time software that is analyzable with state-of-the art static WCET analyzers. However, the set of admissible hardware features is restricted to those features that have been implemented for the model of the Infineon TriCore 1, described in Section 7.2.1. This includes

- parallel execution,
- buffers,
- static branch prediction.

As a consequence of allowing buffers, the considered class of processors exhibits timing anomalies as illustrated by the example of Figure 7.4. More advanced features, like dynamic branch prediction, out-of-order execution, and speculation, can also be modeled in an FSM and therefore, at least in principle, be analyzed using the same method. However, these features tend to significantly increase the size of pipeline models. Getting satisfactory performance with such models probably requires new optimizations. The problem of model size is especially relevant when addressing caches. Section 6.5 discusses the arising problems. The bottom line is that handling caches within the presented symbolic analysis domain is probably infeasible in practice. Section 6.5 discusses an alternative solution that integrates the symbolic pipeline domain with an abstract cache domain.

## 6.2. Basic Symbolic Representation

We compute the abstract semantics of pipeline analysis using a symbolic state traversal algorithm based on image computation (see Definition 4.12). The algorithm explores the state space of a *deterministic* FSM which is specified in terms of its symbolic transition relation as in Definition 4.10. Inputs of the FSM are *not constrained* during the exploration.

In Section 5.2.3, the computation of the abstract semantics of pipeline analysis has been defined using a *non-deterministic* FSM called the abstract pipeline. The inputs of this FSM are obtained from the analyzed program. For the symbolic representation of pipeline domains this non-deterministic FSM is transformed into a deterministic FSM by adding new inputs which enumerate all possible non-deterministic choices. The resulting FSM is called the *abstract pipeline model*. Analyzing a certain program means to restrict the feasible transitions of the model and thereby the reachable states in the state traversal. This restriction is expressed by a *program transition relation*. The process of building this transition relation specifies how an abstract pipeline input relates to certain state variables of the model. The program transition relation forbids model transitions that are infeasible because of program information which also includes the results of preceding analyses, like value analysis. This is equivalent to saying that the program transition relation encodes the program that is interpreted by the abstract pipeline model. The subsequent sections show how the transition relations of the model and the program are defined and constructed.

### 6.2.1. Representation of Pipeline Models

We stated that the abstract pipeline model is a deterministic FSM which is obtained from the non-deterministic FSM of the abstract pipeline by adding new inputs that enumerate all possible non-deterministic choices. This transformation is a concept that explains the relationship between both FSMs. In practice, the deterministic FSM can be specified directly without constructing the non-deterministic FSM first. The non-determinism of the abstract pipeline is modeled by declaring transitions that depend on inputs. Example specifications of abstract pipeline models are shown in Figure 6.3 and Appendix A.1.

**Definition 6.1 (Abstract pipeline model)**
*An abstract pipeline model, or model for short, is a deterministic FSM $M = (\widehat{Q}, I, \delta)$ with $n$ binary state variables and $t$ binary inputs. $\widehat{Q} \subseteq \mathbb{B}^n$ is the set of abstract pipeline states, $I \subseteq \mathbb{B}^t$ is the set of inputs, and $\delta : \widehat{Q} \times I \to \widehat{Q}$ is the next-state function of the model. An abstract state is denoted by a vector of binary state variables $\vec{x} \in \widehat{Q}$ and an input is denoted by a vector $\vec{\imath} \in I$.*

**Note:** *Definition 6.1 redefines the meaning of the symbols $I$ and $\delta$. Neither the set of inputs of the abstract pipeline model nor its next-state function is the same as the set of inputs or the next-state function of the concrete pipeline of Definition 5.3.*

We use Definition 4.9 and Definition 4.10 to obtain symbolic representations for sets of abstract pipeline states and the transition system of the model as Boolean functions. The characteristic function of a set of abstract pipeline states $\widehat{A} \in \widehat{Q}$ is given by:

$$\mathbf{A} : \mathbb{B}^n \to \mathbb{B}$$
$$\mathbf{A}(\vec{x}) = 1 \Leftrightarrow \vec{x} \in \widehat{A}$$

**Definition 6.2 (Model transition relation)**
*Let $M = (\widehat{Q}, I, \delta)$ be an abstract pipeline model. The Boolean function*

$$\begin{aligned} \mathbf{T}_M &: \mathbb{B}^n \times B^t \times \mathbb{B}^n \to \mathbb{B} \\ \mathbf{T}_M(\vec{x}, \vec{\imath}, \vec{y}) &= 1 \Leftrightarrow \delta(\vec{x}, \vec{\imath}) = \vec{y} \end{aligned}$$

*is called the model transition relation of M.*

Reachability analysis as in Equation 4.4 using the model transition relation yields all states of $M$ that are reachable by execution of *any* program.
The model transition relation can be constructed from a specification of the abstract pipeline model in a hardware design language (HDL) like Verilog or VHDL. To this end, the HDL specification is compiled into a netlist that specifies a next-state function $\delta_k$ for each state variable $v_k$ of the model. This operation is a well-known problem in hardware verification and solutions are readily available [Che94]. The transition relation of the variable $v_k$ is then given by the characteristic function of its next-state function:

$$\begin{aligned} \mathbf{T}_k &: \mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B} \to \mathbb{B} \\ \mathbf{T}_k(\vec{x}, \vec{\imath}, y_k) &= 1 \Leftrightarrow \delta_k(\vec{x}, \vec{\imath}) = y_k \end{aligned}$$

The complete model transition relation is obtained by the conjunction of the transition relations of all $n$ state variables:

$$\mathbf{T}_M(\vec{x}, \vec{\imath}, \vec{y}) = \prod_{k=1}^{n} \mathbf{T}_k(\vec{x}, \vec{\imath}, y_k) \tag{6.1}$$

Section 6.2.3 shows an example specification of a model transition and describes how the transition relation of a state variable is constructed in practice.

### 6.2.2. Program Representation

The abstract pipeline of Definition 5.8 consumes inputs that describe program information such as

- instruction types and operands,
- control flow information, such as possible successors for branch instructions, and
- value analysis results, given as ranges, for memory access operations.

The relevant inputs need to be known during the symbolic state traversal. To this end, we encode their effects as a program transition relation that restricts the feasible transitions of the model. We use the program representation of Definition 5.1. All required information is statically available as instruction attributes.

**Definition 6.3 (Program transition relation)**
*Let L be a program and M be an abstract pipeline model with n state variables and t binary inputs. The operational semantics of an instruction $l \in L$ with respect to the model M is given by a Boolean function (instruction transition relation) $\mathbf{T}_l : \mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B}^n \rightarrow \mathbb{B}$ that restricts the feasible transitions in the state space exploration of M:*

$$[\![l]\!] = \mathbf{T}_l(\vec{x}, \vec{\imath}, \vec{y})$$

*The operational semantics of L with respect to the model M is given by the conjunction of the transition relations of all instructions:*

$$[\![L]\!] = \prod_{l \in L} \mathbf{T}_l(\vec{x}, \vec{\imath}, \vec{y}) \tag{6.2}$$

*Equation 6.2 is denoted by the Boolean function $\mathbf{T}_L : \mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B}^n \rightarrow \mathbb{B}$ which we call the program transition relation.*

Reachability analysis as in Equation 4.4 using the symbolic transition relation $\mathbf{T}_M^L = \mathbf{T}_M \cdot \mathbf{T}_L$ yields the set of all abstract pipeline states of $M$ that are reachable by execution of the program $L$. Hence, the symbolic state traversal for WCET analysis of pipeline models can be based on $\mathbf{T}_M^L$.
Building the transition relation $\mathbf{T}_l$ for an instruction $l$ requires knowledge about the design of the abstract pipeline model. In particular, one needs to know in which states certain attributes of $l$ are consumed and how their values relate to certain state variables of the model. In practice, $\mathbf{T}_l$ can be constructed as the conjunction of transition relations that constrain the next-state values of individual state variables and individual instruction attributes. Examples are given in the following section.

### 6.2.3. Generating the Transition Relations

We give two examples to show how the transition relations for individual state variables are generated in practice. The first example shows the construction of the model transition relation using existing tools and methods from hardware model checking. The second example shows the construction of the program transition relation which is specific to our method.

#### Example 1: Model Transition Relation

Figure 6.1 shows Verilog code that uses a binary state variable *exec_is_branch* for propagating the value of an instruction attribute *is_branch* from the fetch unit into the execute stage of a pipeline model. The binary variable *clk* denotes the clock signal. It is used to guard the `always` block that defines transitions on the rising edge of the clock. The code states that the variable *exec_is_branch* is overwritten with the current value of *fetch_is_branch* whenever the *clk* variable has the value 1.
The Verilog code of Figure 6.1 is compiled into the tabular BLIF [BCH+91] representation shown in Figure 6.2. Intermediate variables have been removed to improve

readability. The transition relation of the state variable *exec_is_branch* is given in the form of a truth table with three inputs and a single output. The inputs and output appear in the order that is defined in the *.table* declaration. The default output is 0, which is specified by the `.def` declaration. The truth table states that the next-state value of *exec_is_branch* is 1 if the variables *clk* and *fetch_is_branch* both have the value 1 in the present state, or if the value of *clk* is 0 and *exec_is_branch* has the value 1 in the present state. Otherwise the next-state value of *exec_is_branch* is zero. This is equivalent to the proposition of the Verilog code of Figure 6.1. A corresponding BDD for the truth table can be obtained by the following algorithm:

1. For each entry in the truth table, construct a simple BDD as follows:
   a) Allocate a non-terminal node $n$ and label it with the column name (e.g., $var(n) = clk$).
   b) Connect $high(n)$ with the terminal node that corresponds to the value of the entry (either 1 or 0) and $low(n)$ with the remaining terminal node.
2. Compute the conjunction of the column BDDs for each row.
3. Compute the disjunction of all row BDDs.

The constructed BDD expresses the transition relation of the binary state variable *exec_is_branch*. The complete model transition relation is obtained by application of Equation 6.1 to the transition relations of all state variables. A larger example of Verilog code that describes the fetch unit of a pipeline model is given in Appendix A.1.

### Example 2: Program Transition Relation

Figure 6.3 shows Verilog code for updating the current fetch address in the fetch unit of a pipeline model. The fetch address is stored in the 30-bit[1] variable *fetch_address*. The current fetch address is either incremented by the instruction size, which is 4 bytes in this example, or set to the current value of the program inputs $branch\_target_2$ to $branch\_target_{31}$. The program inputs are referred to using the special syntax `$ND(0,1)` which is part of the VIS[2] [BHSV+96] dialect of Verilog. This syntax states explicitly that the symbolic state traversal does not constrain binary inputs but considers both possible values. The inputs are non-deterministic.
We now create a transition relation for each of the 30 program dependent variables $fetch\_addr_2$ to $fetch\_addr_{31}$. This relation restricts the feasible transitions when executing a branch instruction. The execution of a certain instruction is defined over the state variables $exec\_addr_2$ to $exec\_addr_{31}$ which denote the instruction address that is stored in the execute stage of the pipeline model. Let $l$ be an instruction and

[1]The two least-significant bits can be omitted because we assume that instructions are 4-byte aligned.
[2]VIS stands for Verification Integrated with Synthesis. See Section 7.1 for more information about VIS.

```
always @ (posedge clk) begin
   ...
   exec_is_branch = fetch_is_branch;
   ...
end
always @ (negedge clk) begin
   ...
end
```

**Figure 6.1.:** Verilog code for propagating a binary instruction attribute from the fetch unit into the execute stage of a pipeline model on the rising edge of the clock signal.

```
.table clk fetch_is_branch exec_is_branch -> exec_is_branch'
.def 0
1 1 1 1
1 1 0 1
0 1 1 1
0 0 1 1
```

**Figure 6.2.:** BLIF representation of the single latch update depicted in Figure 6.1. The truth table expresses that the next-state value of *exec_is_branch* corresponds to $(clk \cdot fetch\_is\_branch) + (\overline{clk} \cdot exec\_is\_branch)$.

```
wire branch_target[2:31];
assign branch_target[2]  = $ND(0,1);
...
assign branch_target[31] = $ND(0,1);

always @ (posedge clk) begin
   ...
   if (0 != fetch_addr) begin
      if (exec_is_branch)
         fetch_addr = branch_target;
      else
         fetch_addr = fetch_addr + 4;
   ...
end
```

**Figure 6.3.:** Verilog code for updating the current fetch address in the fetch unit of a pipeline model.

the attribute *addr* denote the address of an instruction as in Definition 5.1. The fact that $l$ is executed is expressed by a condition

$$exec\_cond_l = \prod_{k=2}^{31} exec\_cond_{l,k}$$

where $exec\_cond_{l,k}$ is defined as

$$exec\_cond_{l,k} = \begin{cases} exec\_addr_k \ , & \text{if } l.addr[k] = 1 \\ \overline{exec\_addr_k} \ , & \text{if } l.addr[k] = 0 \end{cases}$$

The execution condition $exec\_cond_l$ is used in the following update condition which further checks the values of the *clk* and the *exec_is_branch* variables. This is the same check that is performed by the pipeline model of Figure 6.3:

$$cond_l = exec\_addr\_cond_l \cdot clk \cdot exec\_is\_branch$$

Let $fetch\_addr'_k$ denote the next-state instance of the variable $fetch\_addr_k$. The instruction attribute *branch_target* denotes the target of a branch instruction. Then, the corresponding transition relation for a branch instruction $l$ is represented by the following Boolean function:

$$update_{l,branch\_target_k} = \begin{cases} cond_l \cdot fetch\_addr'_k + \overline{cond} \ , & \text{if } l.branch\_target[k] = 1 \\ cond_l \cdot \overline{fetch\_addr'_k} + \overline{cond} \ , & \text{if } l.branch\_target[k] = 0 \end{cases}$$

The constructed function denotes a transition relation that restricts the next-state value of $fetch\_addr_k$ to the value of the instruction attribute $l.branch\_target[k]$ in states in which the fetch address is updated with program information and $l$ is in the execute stage of the pipeline model. The disjunction with the negated update condition ensures that the next-state value of this variable is not constrained in other states. The corresponding BDDs are constructed by the following algorithm:

1. Convert the Boolean update function into conjunctive normal form.
2. For each variable, construct a simple BDD:
   a) Allocate a non-terminal node $n$ and label it with the variable name.
   b) Connect $high(n)$ with the terminal node that corresponds to the value of the variable (either 1 or 0) and $low(n)$ with the remaining terminal node.
3. Compute the conjunction of the BDDs for each clause.
4. Compute the disjunction of all clause BDDs.

An instruction transition relation is built by computing the conjunction of many such relations. Finally, the transition relation of the complete program is obtained by combining all instruction relations as in Equation 6.2. An extensive example of code that constructs the program transition relation of the fetch unit model of Appendix A.1 is found in Appendix A.2.

```
1:  $\mathbf{A}_{out} = \mathbf{Empty}$
2:  while $\mathbf{A} \neq \mathbf{Empty}$ do
3:      $\mathbf{A} = \mathbf{Img}(\mathbf{T}, \mathbf{A})$
4:      $\mathbf{A}_{out} = \mathbf{A}_{out} + (\mathbf{A} \cdot \mathbf{R})$
5:      $\mathbf{A} = \mathbf{A} \cdot \overline{\mathbf{R}}$
6:  end while
```

**Figure 6.4.:** Algorithm for computing $\mathbf{A}_{out} = \mathit{tf}_0(\mathbf{T}, \mathbf{A}, \mathbf{R})$.

## 6.3. Symbolic Computation of Abstract Traces

Based on the symbolic representation of abstract pipeline models and programs we present an implementation of pipeline analysis that uses only symbolic computations on BDDs. Thus, it completely avoids the explicit enumeration of states of the abstract pipeline model. Note that the symbolic implementation is semantically equivalent to the explicit state approach but more efficient when considering large sets of reachable pipeline states.

We first present the symbolic computation of abstract traces on the level of basic blocks. This is the core operation of pipeline analysis (see Section 5.2.3). Upper bounds on the execution time are derived from the lengths of the computed traces. We start by defining several helper functions that are used in the algorithm. Let $G = (B, E, n, s)$ be the basic block graph of the program $L$ and $b \in B$ be a basic block. The function $last : B \to L$ returns the last instruction of a basic block:

$$\begin{aligned} &last : B \to L \\ &last(b) = l : l \textit{ is the last instruction in } b \end{aligned}$$

The characteristic function of the set of states in which $last(b)$ was the last instruction to leave the pipeline, is defined as

$$\begin{aligned} &\mathbf{R}_b : \mathbb{B}^n \to \mathbb{B} \\ &\mathbf{R}_b(\vec{x}) = 1 \Leftrightarrow last(b) \textit{ was the last instruction to leave the pipeline} \end{aligned}$$

In order to define $\mathbf{R}_b$ in practice, the last retired instruction has to be stored in the pipeline state. If a pipeline model can retire $n$ instructions in the same cycle all $n$ retired instructions need to be stored. Finally, the empty set of pipeline states is represented by $\mathbf{Empty} : \mathbb{B}^n \to \{0\}$.

Figure 6.4 shows the symbolic implementation of a basic transfer function $\mathit{tf}_0$ which works for basic blocks with a single successor. The analyzed basic block is given implicitly in the retirement function. The signature of the basic transfer function of Figure 6.4 is

$$\mathit{tf}_0 : (\mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B}^n \to \mathbb{B}) \times (\mathbb{B}^n \to \mathbb{B}) \times (\mathbb{B}^n \to \mathbb{B}) \to (\mathbb{B}^n \to \mathbb{B}) \tag{6.3}$$

We call $\mathit{tf}_0$ with the following arguments:

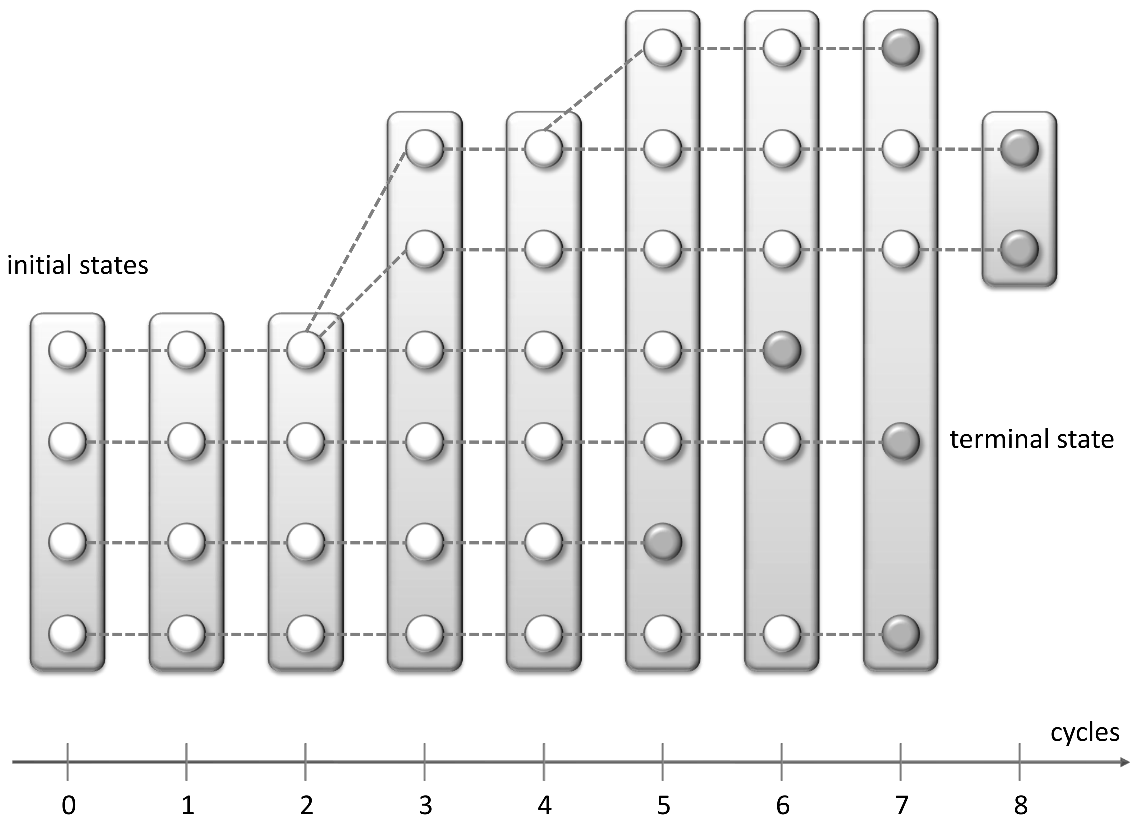


**Figure 6.5.:** Symbolic-state computation of an abstract pipeline trace for a single basic block in BFS order. The abstract trace is computed as a trace of sets (rather than a set of traces as in Figure 5.6).

- the combined transition relation of the pipeline model and the program $\mathbf{T}_M^L = \mathbf{T}_M \cdot \mathbf{T}_L$,
- a set of incoming pipeline states $\widehat{A}$ represented by its characteristic function $\mathbf{A} : \mathbb{B}^n \to \mathbb{B}$, $\mathbf{A}(\vec{x}) = 1 \Leftrightarrow \vec{x} \in \widehat{A}$,
- the set of retiring states $\mathbf{R}_b$ of the current basic block.

The algorithm of Figure 6.4 implicitly constructs the abstract trace $\widehat{exec}(b, \widehat{A})$ for the current basic block $b$ and the incoming set of pipeline states $\widehat{A}$ (see Definition 5.11). In each round of the `while` loop, line 3 of the algorithm computes the successor states in the next cycle. Line 4 adds the retired states to the set of outgoing states $\mathbf{A}_{out}$ and line 5 removes them from $\mathbf{A}$. The algorithm terminates when $\mathbf{A}$ is empty, i.e., all traces ended at a retiring state. At this point, $\mathbf{A}_{out}$ holds all pipeline states that leave $b$. The number of execution cycles of $b$ equals the number of loop iterations.
Figure 6.5 visualizes the implicit computation of an abstract basic block trace by the symbolic algorithm of Figure 6.4. The computation is semantically equivalent to

an explicit-state implementation as depicted in Figure 5.6. However, the symbolic algorithm traverses the state space in breadth-first order. All states that are reached in the same execution cycle with respect to the beginning of the basic block are represented by a single BDD. These states are usually very similar and therefore their BDD representation is much more compact than their explicit enumeration.

### 6.3.1. Integration with a Data Flow Analysis Framework

We discuss the integration of the basic transfer function $tf_0$ of Figure 6.4 with a data flow analysis framework to compute the abstract semantics of pipeline domains as in Section 5.2.3. This requires a fixed point iteration on the CFG of the program under analysis. We observe that a basic block $b$ in the CFG can have more than one successor, e.g., if $b$ contains a conditional branch. This is reflected by the design of the transfer function in Section 3.2 which takes an edge, i.e., a pair of nodes, and a domain element as inputs. The implementation of the basic transfer function $tf_0$ as in Figure 6.4 computes the outgoing states over all outgoing control flow edges. We give an algorithm that uses $tf_0$ to compute the exact set of outgoing states and a more precise execution time bound for a single edge.
Let us assume that if $b$ contains a branch instruction, then this instruction is always $last(b)$.[3] In the case of a conditional branch, we allow for two successor states in the pipeline model (branch taken, not taken). We assume that the pipeline model makes the control decision while decoding $last(b)$. The set of pipeline states decoding $last(b)$ is identified by the following function:

$$\begin{aligned}&\mathbf{D}_b : \mathbb{B}^n \rightarrow \mathbb{B}\\&\mathbf{D}_b(\vec{x}) = 1 \Leftrightarrow \textit{last(b) has been decoded in pre}(\vec{x})\end{aligned}$$

The algorithm that we are about to introduce uses this function for identifying states that have just made a control flow decision. Such a function can always be specified, even for architectures with more complex control flow decisions. If we run $tf_0(\mathbf{T}, \mathbf{A}, \mathbf{D}_b)$, the resulting set $\mathbf{A}_{out}$ contains all states that have decoded $last(b)$ in the last cycle. For an outgoing edge $e$ of block $b$, we characterize the set containing all states that can leave $b$ over $e$ by the function

$$\begin{aligned}&\mathbf{L}_{b,e} : \mathbb{B}^n \rightarrow \mathbb{B}\\&\mathbf{L}_{b,e}(\vec{x}) = 1 \Leftrightarrow \vec{x} \text{ may leave } b \text{ via } e\end{aligned}$$

For example, if $last(b)$ is a conditional branch and $e$ is a true edge, then the conditional branch must have been taken on any state $\vec{x}$ leaving $b$ via $e$. Figure 6.6 shows the improved transfer function $tf^\sharp$ that computes the outgoing set of states for a single control flow edge. Note that the controlling fixed point iteration can be modified to allow caching of the intermediate result – computed in line 1 – for all

[3]This assumption is only introduced for simplifying the description. Architectures featuring delay slots could be handled using a more elaborate $last : B \rightarrow L$ function.

```
1:    $\mathbf{A}_{out} = tf_0(\mathbf{T}, \mathbf{A}, \mathbf{D}_b)$
2:    $\mathbf{A}_{out,e} = tf_0(\mathbf{T}, \mathbf{L}_{b,e} \cdot \mathbf{A}_{out}, \mathbf{R}_b)$
```

**Figure 6.6.:** Algorithm for computing the characteristic function $\mathbf{A}_{out,e}$ of all outgoing pipeline states that may flow over the edge $e$.

outgoing edges of the same block. Let $|a|$ denote the number of loop iterations of algorithm $a$. The number of execution cycles for block $b$ on the path via edge $e$ is then given by the number of loop iterations of the algorithm of Figure 6.6. It is equal to adding up the number of loop iterations in all calls to $tf_0$:

$$|\, tf_0(\mathbf{T}^L_M, \mathbf{A}, \mathbf{D}_b) \,| \; + \; |\, tf_0(\mathbf{T}^L_M, \mathbf{L}_{b,e} \cdot \mathbf{A}_{out}, \mathbf{R}_b) \,| \tag{6.4}$$

We instantiate a data flow analysis framework as follows:

1. The analysis domain is the symbolic domain of abstract pipeline states

$$\mathcal{D}^\sharp = (\{\mathbf{X} : \mathbb{B}^n \to \mathbb{B} \mid \mathbf{X}(\vec{x}) = 1 \Leftrightarrow \vec{x} \in \widehat{X}\}, \\ \mathbf{X} \sqsubseteq \mathbf{Y} \Leftrightarrow \widehat{X} \subseteq \widehat{Y}, +, \cdot, \mathbb{B}^n \to 0, \mathbb{B}^n \to 1)$$

2. The transfer function $tf^\sharp : E \to \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ first computes the functions $\mathbf{D}_b$, $L_{b,\,e}$, and $R_b$ for the incoming edge $e$ with source block $b$. It then calls the algorithm of Figure 6.6 using these functions plus the transition relation $\mathbf{T}^L_M$ and the incoming set of pipeline states $\mathbf{A}$ as arguments. The result is the outgoing set of pipeline states. The number of execution cycles for the set of states that execute $b$ and leave via the edge $e$ is computed as in Equation 6.4.

3. The upper bound operator $\sqcup^\sharp$ is implemented by the disjunction of BDDs, i.e., $X^\sharp \sqcup^\sharp Y^\sharp \equiv \mathbf{X} + \mathbf{Y} \equiv \widehat{X} \cup \widehat{Y}$ for $\widehat{X}, \widehat{Y} \subseteq \widehat{Q}$.

4. The equality test $=^\sharp$ for elements of $\mathcal{D}^\sharp$ is implemented by the equivalence check of BDDs as described in Section 4.2.2.

The presented data flow analysis is fully symbolic. It completely avoids the explicit enumeration of pipeline states and is therefore less affected by the state explosion problem in WCET analysis. Despite the relatively high effort to compute abstract execution traces for small sets of pipeline states, the approach can be expected to perform significantly better when considering larger sets of pipeline states. There are two main reasons for this expectation.

1. The large amount of sharing between pipeline states in the same state exploration layer leads to compact BDDs.

2. Frequently needed operations, like equality checks and computing the union of sets, have very efficient BDD implementations.

However, the efficiency of symbolic pipeline domains depends on the size of the pipeline models. The number of required state variables has a considerable impact on the size of the involved BDDs and therefore the performance of the computation. In order to exploit the advantages of the symbolic representation, it is important to keep the pipeline models as small as possible.

## 6.4. Scaling to Realistic Pipeline Models and Programs

A straightforward implementation of the symbolic approach presented in Section 6.2 and Section 6.3 does not scale to realistic pipeline models and programs. This has two reasons:

- The monolithic transition relation of the pipeline model has a very large BDD representation. It is therefore expensive – and often infeasible – to compute this representation.
- The size of the pipeline model grows with the program size. This is because the model must uniquely identify each instruction in each analysis context. As a result, the performance decreases with growing program size.

To scale, we use three different types of optimizations; all of them improve performance by reducing the sizes of the involved BDDs. The precision of the computed WCET bounds is not affected.

- We rely on standard techniques from model checking to optimize the size of the BDDs that represent the pipeline states and the transition system.
- We use knowledge about the processor to keep the processor model small. We omit information that is not relevant for the timing and statically precompute decisions if possible.
- We exploit the program structure in the symbolic representation of program information and buffer contents in the processor.

The following sections describe each of these optimizations. Since the standard optimizations from model checking have been described in detail in other publications, we focus more on the model optimizations and particularly on how to exploit the program structure. The latter is the key to scale to large programs and highly context-sensitive analyses. The order of the presentation follows the order in which these optimizations are required, i.e., each optimization takes the analysis one step further, to address larger models and programs.

### 6.4.1. Conjunctive Partitioning

Building the transition relation for an FSM as a monolithic BDD by taking the conjunction of the relations for all state variables as in Equation 6.1 is usually infeasible but for the smallest models. Therefore, we represent the model transition relation as a list of partial relations. Each partial relation in the list conjoins the relations for a limited number of variables of the pipeline model. The image computation engine applies the relations one after the other and eliminates variables by quantification in each step to avoid the blowup of the intermediate BDDs during image computation. This is a standard technique known as conjunctive partitioning that we borrowed from model checking. See Section 4.3.2 for a discussion of efficient image computation using a partitioned transition relation.

Conjunctive partitioning can also be applied to the program transition relation since it is also computed as the product of relations for individual variables and instructions (see Equation 6.2). Both partitioned transition relations, for the model and the program, are merged into a single partitioned transition relation for the state traversal. This transition relation can be directly used by the advanced image computation engine described in [RAB+95].

The distribution of the variables over the different partitions and the order in which the partitions are applied affects the speed of image computation. Note that the symbolic representation of Section 6.2 does not enforce any specific distribution and order. In particular, the individual transition relations that represent program information may be conjoined in any order to obtain the program transition relation. The instruction transition relations of Definition 6.3 have been introduced for the sake of a concise and accessible representation. Their intermediate construction is not required in practice.

### 6.4.2. Address and Context Compression

Abstract pipeline models store many program addresses. For example, the modeling of the instruction flow through the pipeline requires state variables to store for each pipeline stage the currently associated instruction. A direct encoding would be to bit-blast these addresses, i.e., a 32-bit address takes 32 state variables. This would be inefficient, since BDD size, and therefore performance, is very sensitive with respect to the number of state variables.

However, a typical program uses only a small fraction of the address space. Exploiting information from the control flow graph, one can *compactly enumerate* all addresses used in the program and then encode these addresses using a number of state variables logarithmic in the size of the set of used addresses. The same optimization can be applied to data addresses that are used by the analyzed program. To this end, the relevant addresses are obtained from the results of the value analysis. Despite the simple principle, address compression – particularly for instruction addresses – is complicated in practice because of two reasons:

- Precise WCET bounds can only be obtained by an interprocedural analysis on the supergraph (see Definition 3.11). To distinguish between executions in different call contexts and loop iterations, pipeline models not only store the addresses of instructions in the various pipeline stages, but also the analysis context in which an instruction has been fetched. The information about analysis contexts is also required to access the context-sensitive results of the value analysis.
- The numbering of program addresses cannot be performed in arbitrary order. It must preserve the order of instruction addresses to allow address arithmetic in the pipeline model. For an example, consider Figure 6.3. If the variable $fetch\_addr$ is not redirected to a branch target, the next fetch address is computed by address arithmetic in the pipeline model.

A naïve handling of many analysis contexts in the BDD representation of a pipeline model would be as inefficient as bit-blasting the program addresses. To avoid this, and to simplify the handling of contexts at the level of pipeline models, we include the information about analysis contexts into the compact address numbering, i.e., we enumerate all program addresses in all analysis contexts. Thus, addresses used by the pipeline model not only identify individual instructions, but also the analysis context in which an instruction has been fetched. If the numbering preserves not only the order of instruction addresses but also the order in which the contexts are analyzed, address arithmetic by the pipeline model then always determines the correct contexts.
The numbering of all program addresses in all analysis contexts is computed by a single pass over the supergraph of the analyzed program. It starts from the analysis entry point and visits all instructions in analysis order. The result is a mapping from addresses in contexts to natural numbers. The construction of the program transition relation uses this mapping to encode the computed numbers instead of program addresses in the symbolic representation. The numbering of all relevant data addresses can be computed by the same pass because the required value analysis results are annotated at the supergraph.

### 6.4.3. Processor-Specific Optimizations

Besides the generic address compression optimization, the size of a pipeline model can be further reduced by exploiting specific properties of the modeled processor pipeline. The abstraction process for obtaining an abstract pipeline model, as described in Section 5.2.2, can already be regarded as an optimization of the model size by omitting information which is not timing-relevant. However, there are two more processor-specific optimizations that can be applied in certain cases:

1. choosing the most compact representation, and
2. statically precomputing information.

A more compact representation is obtained if redundant information is eliminated from the model. This can be achieved by exploiting regularity in the pipeline. Static precomputation of information can be applied in situations where the outcome of a decision can be predicted from the current pipeline state using simple rules. The decision process need not be implemented in the pipeline model. Instead, the possible outcomes are precomputed and added to the program transition relation. We give two examples of processor-specific optimizations for the pipeline model of the Infineon TriCore 1: the compact buffer representation and the precomputation of stall conditions. The examples are meant to illustrate the two optimization principles.

### Compact Buffer Representation

The prefetch buffer of the Infineon TriCore 1 holds up to 8 instructions. Updating buffers and dispatching instructions into the correct pipelines requires type and size information for each instruction. Since the TriCore 1 has two different instruction sizes and two major pipelines, this information can be represented by 2 bits per instruction. We thus represent the timing-relevant buffer contents by 16 bits compared to 16 bytes in the actual processor. This is a classic case of model optimization by omitting irrelevant information. In this case, the actual opcodes and operands of the fetched instructions are not needed for updating the buffer and for correctly dispatching the fetched instructions. If such information is required at a later stage in the pipeline, it can be retrieved from the program representation since each instruction is unambiguously identified by its address and context.
Besides the contents of the prefetch buffer, the pipeline model also needs to identify the addresses that correspond to the current buffer contents. We exploit the update rules for the prefetch buffer to avoid storing the address and context of each contained instruction. Instead, we only store the address of the first instruction in the buffer together with an index that addresses the first instruction that has not been issued into one of the pipelines, yet. Since the size of the buffer is 8 half words, the index can be encoded in only 3 bits. This is an example of a compact representation that is obtained by omitting redundant information. In this case, the redundancy is in the addresses of the instructions in the prefetch buffer.

### Precomputing Stall Conditions

The Infineon TriCore 1 is equipped with two major pipelines that allow for the parallel execution of instructions. The execution in one pipeline may stall because the currently executed instruction depends on the result of another instruction which is still being processed in the other pipeline. The TriCore 1 features a set of rules that define such pipeline stalls in case of unresolved data dependencies. For example, the following code sequence exhibits a write-after-write dependence between the two instructions A and L, because both instructions use the same target register `d0`.

```
add d0, d1, d2  ;  A
ld d0, [a0]0    ;  L
```

| | | | | | |
|---|---|---|---|---|---|
| INT Pipeline | Decode | A | - | | |
| | Execute | | A | - | |
| | Writeback | | | A | - |
| LS Pipeline | Decode | L | L | | |
| | Execute | | - | L | |
| | Writeback | | | - | L |

**Figure 6.7.:** Example of a TriCore 1 pipeline stall. The dependent instructions A and L are issued in parallel, but the LS pipeline stalls for one cycle to resolve the dependence. The symbol '-' in the pipeline diagram denotes nop instructions.

The execution of this code sequence and the effect of the dependency on the TriCore 1 pipeline is depicted in Figure 6.7. Both instructions are issued in parallel, but the LS pipeline stalls for one cycle to resolve the dependence. For our analysis such data dependencies can be precomputed by an interprocedural data flow analysis. The analysis is cheap since it can ignore dependencies that exceed a certain threshold. This threshold can be deduced from the depth of the pipeline as discussed in Section 6.4.4. The results of the data dependency analysis are stored as instruction attributes. In the model transition relation, we use a single program dependent state variable per major pipeline to detect pipeline stalls due to unfulfilled data dependencies. The results of the dependency analysis are encoded in the program transition relation as restrictions for the stall variables. The next-state value of a stall variable in this representation depends on the existence of a data dependence and on the current position of the depending instructions in the pipeline. We give an example, assuming that *stall_ls* is the binary state variable for controlling stalls of the LS pipeline. Attributes of the form

$$[ma|ls]\_[decode|execute|writeback](X)$$

denote that instruction $X$ occupies the decode, execute, or writeback stage of the INT or LS pipeline, respectively. The stall situation depicted in Figure 6.7 can then be handled by adding the following relation:

$$\begin{aligned}
&(stall\_ls' \cdot ma\_decode(A) \cdot ls\_decode(L)) \\
+\ &(\overline{stall\_ls'} \cdot \overline{ma\_decode(A)} \cdot ls\_decode(L)) \qquad (6.5)\\
+\ &\overline{ls\_decode(L)}
\end{aligned}$$

The relation determines the next-state value of the variable *stall_ls* in situations in which instruction $L$ is in the decode stage of the LS pipeline (the first two lines of Equation 6.5). If instruction $A$ is also in the decode stage of the INT pipeline, the stall signal becomes active and causes the LS pipeline to wait until $A$ has proceeded to

the execute stage of the INT pipeline and the dependence is resolved.[4] If instruction $L$ is not in the decode stage of the LS pipeline, the next-state value of *stall_ls* is not constrained (the last line of Equation 6.5). This construction ensures that the conjunction of many such relations does not lead to contradictions, a property which simplifies the construction of the partitioned transition relation of Section 6.4.1. In this example, the instruction $L$ is regarded as the current instruction for which program relations are created. Finally, note that the value of *stall_ls* is not constrained if the decode stage of the LS pipeline does not hold a valid instruction from the analyzed program, e.g., when starting the analysis. However, in such cases we use a special 0-instruction which is treated specifically by the pipeline model. An example of the special treatment of the 0-instruction can also be found in Figure 6.3.
The described optimization requires significantly fewer state variables than explicitly checking for data dependencies in the model. The latter would need additional variables for storing the involved operands.

### 6.4.4. Program Decomposition

The aforementioned optimizations make the construction of the symbolic representations feasible. However, the required number of state variables still heavily depends on the size of the program because all instructions must be uniquely identified in the pipeline model. Additionally, context-sensitive analysis, which is indispensable to obtain sufficient analysis precision, increases the number of individual instructions even further by virtual inlining and unrolling (see Section 3.3). Since BDD performance depends on the number of state variables, the performance of *every single* analysis operation depends on the size of the analyzed program and on the number of analysis contexts. This undesired dependence can be removed by an optimization that is based on two observations:

**Observation 1** *There is an upper bound on the number of instructions that a pipeline (or abstract pipeline model) can process concurrently due to parallel execution, prefetching and speculation.*

**Observation 2** *Pipelines perform out-of-order execution and yet guarantee in-order completion, i.e. even if some later instruction $l_2$ can be executed before some other instruction $l_1$, it will not leave the pipeline before $l_1$.*

To remove the undesired dependence of the state traversal complexity on the program size, we use both observations to limit the range of program information that is required to compute the abstract trace at any program point. We introduce the notion of relevant instructions to describe this range.

[4]The TriCore 1 model uses two transitions per cycle. The two half-cycles correspond to the rising and falling edge of the clock signal. This modeling allows setting and reading a variable in the same cycle. Hence, the model is able to stall in the same cycle in which the dependence is detected.

**Note:** *From now on the term instruction refers to an instruction in a specific analysis context, i.e., an instruction is no longer identified by its address only, but by its address* and *analysis context.*

**Definition 6.4 (Relevant instructions)**
*The set of instructions that is required to compute the abstract execution of a basic block b in a program L is called the set of relevant instructions of b in L.*

All instructions that are contained in a basic block $b$ are trivially relevant for $b$. However, due to the overlapping of instruction execution in the pipeline, also instructions that are not in $b$ can be relevant.

**Definition 6.5 (Overlap bound)**
*Let M be a pipeline model. For a basic block b of a program L, the number of relevant instructions of b in L that are not contained in b is called the overlap of b in L. The overlap bound of M is the maximum of the overlaps of all basic blocks in all programs.*

The overlap bound is a model-specific constant. It expresses the relevant overlapping between a basic block with its neighbors in the control flow graph. We define several helper functions to compute the relevant range for a given program point (identified by an instruction) and overlap bound. The following function determines the distance between two instructions in control flow backward order:

$$pre : L \times L \to \mathbb{N}$$
$$pre(l_i, l_j) = \begin{cases} \min\{k \mid l_j \text{ is the } k\text{-th predecessor of } l_i\}, & \text{if this set is not empty} \\ 0, & \text{otherwise} \end{cases}$$

To compute this function, we traverse the control flow graph backwards in BFS order, starting from the current instruction $l_i$ and stopping when we reached the predecessor $l_j$. When the BFS search stops the current depth equals the distance $k$. The same approach can be used to determine the distance in control flow forward order:

$$post : L \times L \to \mathbb{N}$$
$$post(l_i, l_j) = \begin{cases} \min\{k \mid l_j \text{ is the } k\text{-th successor of } l_i\}, & \text{if this set is not empty} \\ 0, & \text{otherwise} \end{cases}$$

Note that if $l_i$ and $l_j$ are in the same subsuming context of a loop, it is possible to compute both, a pre- and a post-distance, between $l_i$ and $l_j$. In such cases $l_j$ is a predecessor *and* a successor of $l_i$. Based on the pre-distance function, the set of all predecessor instructions within a given range is computed by the following function:

$$\begin{aligned} &preds : L \times \mathbb{N} \to 2^L \\ &preds(l_i, k) = \{l_j : 0 < pre(l_i, l_j) \leq k\} \end{aligned} \tag{6.6}$$

Again, this function is implemented by a traversal over the control flow graph backwards in BFS order starting at $l_i$ and stopping at depth $k$. A similar function computes the set of all successor instructions within a given range:

$$\begin{aligned} &succs : L \times \mathbb{N} \to 2^L \\ &succs(l_i, k) = \{l_j : 0 < post(l_i, l_j) \leq k\} \end{aligned} \tag{6.7}$$

The following theorem establishes an upper bound on the overlap bound of any pipeline model.

**Theorem 6.1 (Maximum overlap bound)**
*Let M be an abstract pipeline model that can hold at most m instructions at the same time and let b be a basic block. The first and the last instructions of b are denoted by $first(b)$ and $last(b)$. The maximum overlap bound of M is $\bar{c} = 2m$ and the set of relevant instructions of b is bounded by*

$$b \cup preds(first(b), \frac{\bar{c}}{2}) \cup succs(last(b), \frac{\bar{c}}{2})$$

**Proof** Let us assume that $b$ contains only a single instruction $l_i$. Then we have to prove that the set of relevant instructions is bounded by $\{l_i\} \cup preds(l_i, m) \cup succs(l_i, m)$. We prove 2 cases:

1. *Only instructions from $preds(l_i, m)$ are in the pipeline when $l_i$ enters it:*
   When $l_i$ enters the pipeline, there can be at most $m$ other instructions in the pipeline. Let $l_j$ be an instruction such that $pre(l_i, l_j) > m$. If $l_j$ is still in the pipeline when $l_i$ enters it, then either there are more than $m$ instructions in the pipeline, or at least one instruction between $l_i$ and $l_j$ that has retired before $l_j$. Both possibilities contradict our observations.

2. *Only instructions from $succs(l_i, m)$ are in the pipeline when $l_i$ retires:*
   When $l_i$ leaves the pipeline, there can also be at most $m$ other instructions in the pipeline. Let $l_j$ be an instruction such that $post(l_i, l_j) > m$. If $l_j$ already entered the pipeline before $l_i$ retires, then either there are more than $m$ instructions in the pipeline, or at least one instruction between $l_i$ and $l_j$ has not been fetched. Again, both possibilities contradict our observations.

Since only instructions from $\{l_i\} \cup preds(l_i, m) \cup succs(l_i, m)$ can be in the pipeline at block $b$, the model $M$ cannot access information related to any other instruction. Hence, such information cannot be required for the state traversal for computing the abstract trace of $b$. The proof trivially holds also for blocks that contain more than one instruction. ■

The maximum overlap bound of a given pipeline model can be determined by counting the state variables that may contain instructions. The actual bound may

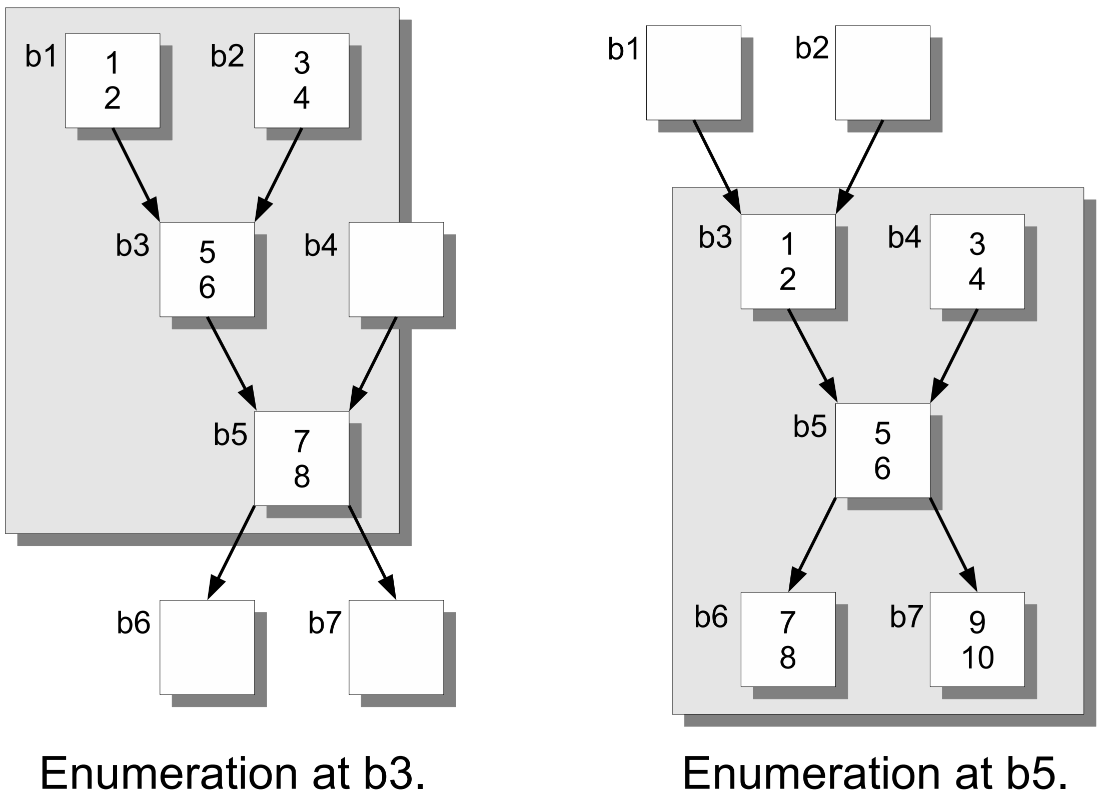


**Figure 6.8.:** Illustration of program decomposition under the assumption that $\bar{c} = 4$. Both cases show the same section of the same basic block graph. At the left hand side we enumerate only instructions that are relevant for the state traversal at block 3. At the right hand side we do the same for block 5.

be smaller if the update logic forbids that certain variables are used concurrently. Theorem 6.1 can be used to obtain a more compact and therefore more efficient representation of instruction addresses and analysis contexts that does not depend on the size of the analyzed program. The idea is that for each basic block we enumerate (see Section 6.4.2) only the addresses and contexts that are relevant for this block. As a result, the required number of state variables for the unambiguous identification of instructions during state traversal is reduced. Let $m$ be the number of instructions that a given pipeline model can hold. Furthermore, $n$ is the number of all instructions in all analysis contexts of the analyzed program. The number of binary state variables that is required for the unambiguous enumeration of all instructions is

$$m \cdot log_2(n) \tag{6.8}$$

For a given basic block, the number of required state variables can be reduced by application of Theorem 6.1 to

$$m \cdot log_2(|b \cup preds(first(b), m) \cup succs(last(b), m)|) \tag{6.9}$$

The result of this equation depends on the number of instructions in the basic block $b$ and on the number of predecessors and successors of $b$ in the control flow graph. In general, the result of Equation 6.9 is much smaller than the result of Equation 6.8. In the following, we denote this compaction of the symbolic representation at a given basic block $b$ by $\lceil\,\rceil_b$. Figure 6.8 shows an example for two blocks $b3$ and $b5$, assuming that $\bar{c} = 4$. The numbers in the blocks correspond to the enumeration of the relevant instructions at $b3$ and $b5$ respectively.

### Symbolic Translation Between Basic Blocks

As a consequence of enumerating only the relevant instructions for each block, the encoding of instructions is no longer globally the same for all blocks of the analyzed program. As an example, consider block 3 in Figure 6.8. When analyzing block 3, the two instructions contained in this block correspond to the numbers 5 and 6. On the other hand, when analyzing block 5, the same instructions correspond to the numbers 1 and 2. The instruction encoding must therefore be translated before propagating states along control flow edges. According to Theorem 6.1 the number of overlapping instructions between two adjacent blocks is bounded by the processor specific overlap bound $\bar{c}$. In the example of Figure 6.8, the translation of the instruction numbering between block 3 and block 5 can be expressed by the following relation between the 4 overlapping instructions:

$$overlap(b3, b5) = \{(5,1), (6,2), (7,5), (8,6)\} \tag{6.10}$$

Such mapping relations between different instruction numbers at neighboring basic blocks can be represented symbolically by BDDs. The relations are expressed over the variables of the abstract pipeline model. For example, the characteristic function of the overlap relation of Equation 6.10 can be specified as

$$\begin{aligned} \mathbf{T}_{b5}^{b3}(\vec{x}, \vec{\imath}, \vec{y}) = 1 \Leftrightarrow \forall (l_i, l_j) \in overlap(b3, b5) :& \\ &\textit{references of } l_i \textit{ in } \vec{x} \textit{ are replaced by references of } l_j \textit{ in } \vec{y} \\ &\wedge \vec{x}, \vec{y} \textit{ are otherwise equivalent} \end{aligned}$$

The example shows that one can specify relations between states of the model which directly express the translation of the instruction enumeration between basic blocks. The practical construction of such relations requires the overlap mapping between two basic blocks and information about which model variables reference instructions and how the instruction numbers are encoded. Translating sets of abstract pipeline states between $p$ and $s$ can then be implemented symbolically by computing the image $\mathbf{A}'$ of a set of pipeline states $\mathbf{A}$ over the mapping relation $\mathbf{T}_s^p$

```
1:    $\lceil \mathbf{A}_{out} \rceil_p = tf_0(\lceil \mathbf{T} \rceil_p, \lceil \mathbf{A} \rceil_p, \lceil \mathbf{D}_p \rceil_p)$
2:    $\lceil \mathbf{A}_{out,e} \rceil_p = tf_0(\lceil \mathbf{T} \rceil_p, \lceil \mathbf{L}_{s,e} \rceil_p \cdot \lceil \mathbf{A}_{out} \rceil_p, \lceil \mathbf{R}_p \rceil_p)$
3:    $\lceil \mathbf{A}_{out,e} \rceil_s = \mathbf{Img}(\mathbf{T}_s^p, \lceil \mathbf{A}_{out,e} \rceil_p)$
```

**Figure 6.9.:** Algorithm for computing the outgoing set of pipeline states, translated into the target block instruction numbering. The blocks $p$ and $s$ are the source and target blocks of the edge $e$, i.e., $(p, s) = e$.

as $\mathbf{A}' = \mathbf{Img}(\mathbf{T}_s^p, \mathbf{A})$. We include this translation into the algorithm of Figure 6.6 to obtain the optimized transfer function that is depicted in Figure 6.9. Abstract basic block execution for a compatibly encoded set of states $\lceil \mathbf{A} \rceil_b$ can now be performed more efficiently by application of the improved algorithm. Note that for a block $b$ with several predecessors, all incoming states are translated into the same range $\lceil\, \rceil_b$. Therefore, the instruction numbering is consistent over all incoming edges. Incoming sets of states from the predecessor blocks can be safely combined by computing the disjunction of the Boolean functions before proceeding with the state traversal at $b$. The additional cost for translating between different basic blocks is amortized by the savings achieved by reducing BDD size during state traversal. Moreover, $\lceil \mathbf{T} \rceil_b$ conjoins fewer relations than $\mathbf{T}_L$ which further improves the performance of image computations.

## 6.5. Interfacing Abstract Caches

The presented approach for symbolic pipeline analysis cooperates with the static WCET analysis framework by exchanging analysis results, e.g., with control flow and value analysis. A commonality of these analyses is the fact that they run prior to pipeline analysis. Hence, cooperation boils down to importing statically available analysis results. In contrast, the abstract interpretation of caches [Fer97] cannot be separated from pipeline analysis. The cache state depends on the order of memory accesses and therefore on the state of the pipeline. The pipeline state in turn is influenced by the latency of instruction and data fetches which depends on the cache state. Explicit-state implementations of pipeline analysis establish a one-to-one relationship between pipeline and cache states, i.e., they combine each abstract cache state with a single abstract pipeline state. The pipeline state triggers an update of its associated cache state whenever the processor accesses a cached memory area [FHL+01]. The presented symbolic-state approach cannot afford a one-to-one combination of pipeline and cache states without losing the advantages of symbolic state space exploration. We describe a semi-symbolic domain that efficiently integrates abstract interpretation based cache analysis with our symbolic pipeline analysis while preserving a high analysis precision.

### 6.5.1. The Interface Problem

Cache analysis [Fer97] operates on abstract representations of cache states. The abstract representation allows to trade precision for efficiency. Soundness is maintained by losing information only on the safe side, i.e., the result over-approximates the concrete cache states but it never misses a reachable cache state. The interface of the cache analysis comprises functions to query and update abstract caches with intervals of memory addresses. It also features a join operator for joining two cache states into another cache state that over-approximates both. The join operation may lose precision. There are two possibilities for interfacing caches with our symbolic pipeline analysis:

1. Including the cache into the symbolic representation of pipeline states.

2. Associating an abstract cache representation with a symbolic representation of pipeline states.

Let us consider the first approach. The representation of a cache requires too many state bits to allow for a straightforward BDD representation. For example, consider a 2-way set-associative cache with 128 sets and 32 bytes line size. Representing the state of such a cache requires a prohibitive number of 5120 state bits.[5] Even if we use address compression (Section 6.4.2) the number of bits can only be reduced to $128 \cdot 2 \cdot log_2(|\{Addresses\}|)$ which is still very large compared to the size of an abstract pipeline model (see Section 7.2.3).
The second possibility – associating an abstract cache representation with a symbolic representation of pipeline states – seems equivalent to the approach that is taken by explicit-state implementations. However, symbolic pipeline analysis cannot afford a one-to-one combination of pipeline and cache states without losing the advantages of symbolic state space exploration. The explicit handling of caches would require the same explicit enumeration of pipeline states that the symbolic representation is trying to avoid. The next section presents a domain that is based on this second possibility but maintains a more favorable combination of pipeline and cache states.

### 6.5.2. A Semi-Symbolic Domain for Microarchitectural Analysis

We describe a semi-symbolic domain that integrates an abstract cache representation with a symbolic representation of pipeline states. The explicit enumeration problem is avoided by maintaining an efficient relationship between pipeline and cache states. The basic idea is that we combine a set of pipeline states (represented symbolically by a BDD) with a single abstract cache state. The product of the pipeline and cache domains is thus based on an *n-to-one* combination. This allows us to preserve

[5]The considered cache has 256 lines (2 ways times 128 sets) and we represent its state by identifying the memory block that each line holds. Of the memory block address, the lower 12 bits can be omitted because 5 bits identify the byte address in the cache line and 7 bits locate the cache set and are thus redundant. Hence, we have $256 \cdot (32 - 12) = 5120$.

the benefits of the symbolic representation by manipulating sets of pipeline states symbolically. Elements of this product are called *partitions* of hardware states. The analysis maintains sets of such partitions.

**Definition 6.6 (Partition)**
*Let $\mathcal{D}^\sharp$ and $\mathcal{D}^{\widehat{c}}$ denote the symbolic pipeline domain and the abstract cache domain, respectively. A partition of abstract hardware states is a tuple of type $\mathcal{D}^\sharp \times \mathcal{D}^{\widehat{c}}$. We denote the set of all partitions by $H$ .*

The pipeline and cache domains in the above definition feature the usual comparison and join operators for complete lattices (see Section 3.1). The symbolic pipeline domain is defined as in Section 6.3.1 and the implementation of the cache domain is opaque. The semi-symbolic domain for the combined pipeline and cache analysis is based on the power set of $H$:

$$\mathcal{D}^h = (2^H, \cup, \cap, \emptyset, H)$$

The term partition of Definition 6.6 refers to the fact that for any domain element, each abstract hardware state is represented by exactly one pipeline-cache tuple, i.e., a partition. This is guaranteed by the construction of operations on domain elements which is detailed in the subsequent sections.

### Updating Partitions of Abstract Hardware States

First, we show the update of a single partition $(X^\sharp, A^{\widehat{c}}) \in H$. Let $\mathcal{A}_C$ be the set of all addresses in cached memory[6] that are accessed by the analyzed program. The pipeline model then needs $m = log_2(|\mathcal{A}_C|)$ binary state variables for addressing memory. We require that these variables appear first in the BDD representation. In the following, we define several functions for operations on partitions. Let $\mathbb{I} \subset \mathbb{N} \times \mathbb{N}$ denote the set of intervals such that $\forall (l, u) \in \mathbb{I} : l \leq u$. The addressed interval can be obtained by a function

$$acc : \mathcal{D}^\sharp \to \mathbb{I}$$

that inspects the first $m$ BDD variables. We defer the discussion of its efficient implementation to Section 6.5.3. For a given access interval and an abstract cache state, the classification function of the cache domain determines whether an access results in a cache hit $(0, 1)$ or miss $(1, 0)$. The result of this query can also be *undecided* $(1, 1)$ if precise information has been lost due to abstraction or if the

[6]For the remainder of this chapter we assume that *all* memory accesses address cached memory regions.

interval comprises both, cache hits and misses.

$$cl : \mathcal{D}^{\widehat{c}} \times \mathbb{I} \to \{(0,1),(1,0),(1,1)\}$$

$$cl(A^{\widehat{c}},[l,u]) = \begin{cases} (0,1) & [l,u] \text{ surely hits } A^{\widehat{c}} \\ (1,0) & [l,u] \text{ surely misses } A^{\widehat{c}} \\ (1,1) & [l,u] \text{ may hit and may miss } A^{\widehat{c}} \end{cases}$$

For a pipeline model with $n$ binary state variables and $t$ binary inputs, the result of $cl(A^{\widehat{c}}, acc(X^{\sharp}))$ can be encoded as a symbolic cache transition relation $\mathbf{T}_C$ by the following encoding function:

$$enc : \mathbb{B}^2 \to (\mathbb{B}^n \times \mathbb{B}^t \times \mathbb{B}^n \to \mathbb{B})$$

The computed cache transition relation restricts the possible transitions of the model relation $\mathbf{T}_M$. It only allows for transitions that correspond to the result of the cache query. This is analogue to the construction of $\mathbf{T}_L$ from statically available program information. We now have all the functions for implementing the exchange of information between both domains. Let us consider the actual update of a partition. To this end, we define an opaque update function for abstract caches:

$$up^{\widehat{c}} : \mathcal{D}^{\widehat{c}} \times \mathbb{I} \to \mathcal{D}^{\widehat{c}}$$

The update of a single partition $(X^{\sharp}, A^{\widehat{c}})$ is then computed by a function $up^H : H \to H$. The implementation of this function is depicted in Figure 6.10. It first determines the interval $I$ of memory addresses that is accessed by the pipeline states in $X^{\sharp}$. It then queries the cache domain to determine whether the access hits or misses the cache and – based on this information – constructs the BDD $\mathbf{T}_C$ for restricting the reachable pipeline states. The constructed BDD is conjoined with the BDDs $\mathbf{T}_M$ and $\mathbf{T}_L$ to obtain the effective transition relation for the next update. By application of the image operator on the computed transition relation and the set of pipeline states $X^{\sharp}$, it computes the set of successor pipeline states. The next cache state is obtained by application of the cache domain update function on the current cache $A^{\widehat{c}}$ and the accessed interval $I$.

### Balancing Pipeline and Cache States

In order to maintain a favorable n-to-one combination of pipeline and cache states, we introduce a balancing operation to be applied in each round of the state traversal. The balancing operation involves two steps: *partitioning* and *join*.

**Partitioning.** The partitioning step is based on the decomposition of the BDD of pipeline states using Shannon's expansion (Theorem 4.1). Let $(X^{\sharp}, A^{\widehat{c}}) \in H$ be a partition of abstract hardware states. The first $m$ state variables in $X^{\sharp}$ encode the

```
$up^H(X^\sharp, A^{\hat{c}}) =$
        let $I = acc(X^\sharp)$ in
        let $\mathbf{T}_C = enc(cl(A^{\hat{c}}, I))$ in
            $(\mathbf{Img}(\mathbf{T}_M \cdot \mathbf{T}_L \cdot \mathbf{T}_C, \mathbf{X}^\sharp),\ up^{\hat{c}}(A^{\hat{c}}, I))$
```

**Figure 6.10.:** Implementation of the update function $up^H : H \to H$.

accessed interval of memory addresses. We partition $(X^\sharp, A^{\hat{c}})$ by a function that recursively decomposes $X^\sharp$ into its cofactors with respect to the first $m$ state variables:

$$part : H \to \mathcal{D}^h$$

A new partition is created for each of the final cofactors together with a copy[7] of the cache state $A^{\hat{c}}$. As a result of this operation, all pipeline states in a new partition $(X_a^\sharp, A^{\hat{c}}) \in part(X^\sharp, A^{\hat{c}})$ access the same interval of memory addresses. Note that $up^H(X_a^\sharp, A^{\hat{c}})$ yields a more precise successor cache state than $up^H(X^\sharp, A^{\hat{c}})$.
The worst-case complexity of the full recursive decomposition over all $m$ state bits is in $O(2^m)$. It is particularly expensive for large intervals because it enumerates all potentially accessed memory addresses. However, when considering instruction fetches or small intervals of data addresses (which correspond to precise value analysis results), many cofactors are constant zero and can be dropped immediately. Furthermore, the decomposition can be simplified by exploiting knowledge about relevant[8] program information. For example, assume that all pipeline states at a certain program point are known to access one of the two disjoint intervals `[0x100,0x1ff]` or `[0x200,0x2ff]`. These two intervals can be encoded with $m = 10$ bits $x_0, \ldots, x_9$. Computing just the positive and negative cofactors of the most significant variable $x_0$ directly yields the partitions that correspond to these two intervals. For overlapping intervals the decomposition can also be restricted to the most significant $k < m$ variables. The access intervals of the resulting partitions overestimate the information obtained from the value analysis. Here, the semi-symbolic domain allows us to trade precision for computability.

**Join.** Excessive partitioning might lead us back to the explicit enumeration problem. In the worst case, each partition in a domain element $D^h \in \mathcal{D}^h$ encodes only a single pipeline state. We prevent this by applying a join operator to partitions of $D^h$. Two partitions $(X^\sharp, A^{\hat{c}})$ and $(Y^\sharp, B^{\hat{c}})$ are joined by an operator $\sqcup^h$ which performs a pairwise join operation on the elements of the partitions:

[7]Implementations can use references to improve performance.
[8]The meaning of the notion *relevant* is the same as in Definition 6.4.

$$(X^\sharp, \widehat{A^c}) \sqcup^h (Y^\sharp, \widehat{B^c}) = (X^\sharp \sqcup^\sharp Y^\sharp, \widehat{A^c} \,\widehat{\sqcup^c}\, \widehat{B^c})$$

To minimize the loss of cache precision, we join only partitions whose pipeline states access the same interval of memory addresses. This restriction also prevents us from undoing the partitioning. The loss in cache precision can be limited further by joining only hardware states with similar caches. This however requires a similarity metric for abstract cache states. A simple but efficient similarity metric is, to only join two cache states $\widehat{A^c}, \widehat{B^c} \in \widehat{\mathcal{D}^c}$ if one of them already over-approximates the other, which is equivalent to

$$\widehat{A^c} \,\widehat{\sqcup^c}\, \widehat{B^c} = \widehat{A^c} \quad \text{or} \quad \widehat{A^c} \,\widehat{\sqcup^c}\, \widehat{B^c} = \widehat{B^c}$$

Besides balancing the relationship between pipeline and cache states and optimizing the representation for an efficient implementation of the function $acc : \mathcal{D}^\sharp \to \mathbb{I}$, the application of regular partitioning and join operators also ensures a canonical representation; because of the join operation, a particular abstract hardware state always ends up in exactly one partition of an element of $\mathcal{D}^h$.[9] This property allows for an efficient equality check of data flow elements by pairwise invocation of the equality operators of the two underlying domains on the contained partitions. It is most efficient if the number of partitions is small.

### 6.5.3. State Traversal and Performance

The state traversal for micro-architectural analysis on the domain $\mathcal{D}^h$ is implemented by repeated application of the function $up^H : H \to H$ to all elements of a domain element $D^h \in \mathcal{D}^h$. Partition and join functions are applied in each round of the traversal for balancing pipeline and cache states before starting the next round. Hence, the chained execution of

1. $up^H$ over all partitions,
2. *partition*, and
3. *join*

gives the transfer function $tf^H$ of the domain $\mathcal{D}^h$. An example update using this transfer function is depicted in Figure 6.11. The domain is most efficient if each cache state is associated with a large number of pipeline states. This allows for a small number of BDD operations which exploits the caching of intermediate results that is typical for BDD algorithms. Moreover, it significantly reduces the required number of cache updates since we perform a single cache update for all of the associated pipeline states. Note that a small number of partitions per domain element is also

[9] After the join, states in two different partitions differ at least in their assigned memory access intervals.

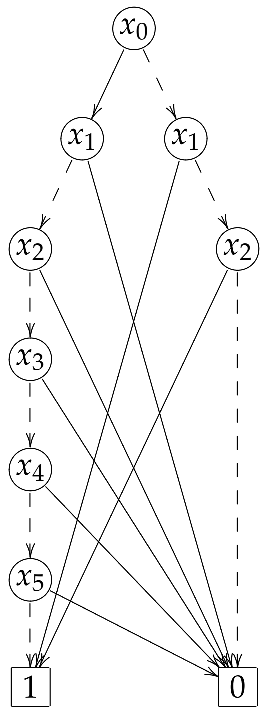
$$\begin{tikzcd}
(X^\sharp & \arrow[d, Rightarrow, "up^H"'] & A^{\hat{c}}) & (Y^\sharp & \arrow[d, Rightarrow] & B^{\hat{c}}) & (Z^\sharp & \arrow[d, Rightarrow] & C^{\hat{c}}) \\
(X^\sharp_1 \arrow[d, "partition"'] & & A^{\hat{c}}_1) \arrow[dd, dashed, "join"'] & (Y^\sharp_1 \arrow[d] \arrow[drr] & & B^{\hat{c}}_1) \arrow[ddlll, dashed] \arrow[ddrrr, dashed] & (Z^\sharp_1 \arrow[dr] & & C^{\hat{c}}_1) \arrow[dd, dashed] \\
X^\sharp_{11} \arrow[d, "join"'] & & & Y^\sharp_{11} \arrow[dlll] & & Y^\sharp_{12} \arrow[dr] & & Z^\sharp_{11} \arrow[dl] & \\
(X^\sharp_{11} \sqcup^\sharp Y^\sharp_{11} & & A^{\hat{c}}_1 \sqcup^{\hat{c}} B^{\hat{c}}_1) & & & & (Y^\sharp_{12} \sqcup^\sharp Z^\sharp_{11} & & B^{\hat{c}}_1 \sqcup^{\hat{c}} C^{\hat{c}}_1)
\end{tikzcd}$$

**Figure 6.11.:** Example of applying the transfer function to a domain element $D^h \in \mathcal{D}^h$ containing 3 partitions. The 3 steps of the transfer function are annotated at the left-most edges which represent operations from the respective step.

desirable. Numbers for assessing the expected number of partitions and the expected relations between pipeline and cache states in practice are given in Section 7.2.5.
A favorable combination of pipeline and cache states is maintained by the regular application of the join operator. The prior application of the partitioning operator minimizes the loss of cache precision and optimizes the BDD representation to allow for an efficient implementation of the function $acc : \mathcal{D}^\sharp \to \mathbb{I}$. Its efficiency depends on the fact that

- the variables for addressing memory appear first in the BDD, and
- all encoded pipeline states are assumed to access the same interval of addresses.

Hence, it suffices to enumerate the satisfying paths over the BDD nodes that correspond to the first $m$ variables. In this case, a path is satisfying if it ends at a non-terminal node that is labeled $x_k$ with $k > m$, i.e., it does not end at the terminal node 0 and consequently the terminal node 1 is reachable via the variables that represent the pipeline states. Note that we do *not* enumerate all satisfying assignments of the first $m$ variables. Variables that are implicitly represented by dont-care nodes are ignored. Let us consider the example depicted in Figure 6.12. The example BDD shows only the first 6 state variables for accessing memory, i.e., we have $m = 6$. Note that in the full representation, the terminal node 1 would be replaced by a subgraph that represents the set of associated pipeline states. The satisfying paths over the example BDD of Figure 6.12 are depicted in the first table of Figure 6.12. To determine the interval that corresponds to a satisfying path, we set all dont-care nodes to 0 to obtain the lower bound (see table 3 in Figure 6.12), and to 1 to obtain



| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | - | - | - | - |
| 0 | 0 | 1 | - | - | - |

1. satisfying paths

| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | ub |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 32 |
| 0 | 1 | 1 | 1 | 1 | 1 | 31 |
| 0 | 0 | 1 | 1 | 1 | 1 | 15 |

2. upper bound

| $x_0$ | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | lb |
|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | 32 |
| 0 | 1 | 0 | 0 | 0 | 0 | 16 |
| 0 | 0 | 1 | 0 | 0 | 0 | 8 |

3. lower bound

**Figure 6.12.:** BDD representation of the memory access interval $[8, 32]$. In the full representation, the terminal node 1 is replaced by a subgraph that represents the set of associated pipeline states. The tables on the right-hand side show the computation of the lower and upper bounds of the intervals that correspond to the satisfying paths. The complete interval is then computed as $[\min\{32, 16, 8\}, \max\{32, 31, 15\}] = [8, 32]$.

the upper bound (see table 2 in Figure 6.12). Finally, we obtain the represented interval by taking the minimum and maximum of the intervals over all satisfying paths.

The example shows that the interval can be computed from the BDD without enumerating all contained addresses. Note that the computational effort does not grow significantly if the interval shares a larger address prefix (using additional state variables $x_6, x_7, \ldots, x_m$ to address memory). The additional state variables either allow only for a single assignment, or most of them are dont-care nodes. The number of satisfying paths in the BDD can be expected to stay small. In particular, it will remain much smaller than the number of contained addresses.

## 6.6. Summary

The presented symbolic pipeline domain completely avoids the explicit enumeration of reachable pipeline states. It is therefore more efficient than previous approaches when covering substantial subsets of the state space of a pipeline model. The effort for computing cycle updates is independent of the size of the analyzed program and of the number of states reached by the analysis. Instead it depends on the size of the BDD representations, the most important contributing factor being the size of the pipeline model itself. In addition to efficiently handling large sets of reachable pipeline states, the approach also allows to efficiently join domain elements, e.g., in cases of joining control flow. Join operations are particularly awkward in existing explicit-state implementations because they require pairwise comparison of all pipeline states in all sets. Also, the result of the join needs as much space (in terms of memory) as all joined values together. The corresponding symbolic operations on BDDs in symbolic pipeline analysis are more efficient in both respects: computational effort and size of the result.
In combination with the cache domain, the symbolic approach to pipeline analysis also provides a very desirable feature that is not available with previous methods: it allows to choose between a more precise result and a faster analysis at run-time. Since the combined domain $\mathcal{D}^h$ is a power set of pairs of BDDs and abstract cache states, we have some degree of freedom when partitioning and joining elements. If we aggressively join BDDs and abstract cache states, we lose cache precision but we get the most sharing between similar pipeline states and reduce the number of operations for updates. On the other hand, if we avoid such joins the analysis may produce more precise results at the price of higher memory consumption and computation time. Similarly, aggressive partitioning reduces the sharing and improves the precision. This allows the development of new optimization strategies for fine-grained balancing of efficiency versus precision.
Finally, we would like to point out an advantage that we have not explicitly mentioned yet. Because of the straightforward correspondence between FSMs and BDDs (by representation of the FSMs transition relation, see Definition 4.10 and Figure 4.4) it is relatively easy to generate parts of the implementation of a pipeline domain from a model specification in VHDL or Verilog or even a graphical definition language. One of the reasons is that the symbolic approach clearly separates the model specification (symbolic transition relation) from the state traversal (image computation). Also, tools for compiling such specifications into symbolic transition relations have already been developed in the context of symbolic model checking, e.g., in the VIS system or in Uppaal, and can be re-used for pipeline analysis. Implementing similar compilers for explicit-state pipeline analyses is not easy because the state traversal algorithm is entwined with the transition function in order to handle the non-determinism. However, this problem has already been tackled and a working implementation of such a compiler is reported in [SP10].
Despite all of these advantages, symbolic pipeline domains also have a few drawbacks that cannot be ignored. The size of the pipeline models is critical and careful tuning

is needed to cut down the number of state bits required to represent a pipeline state. This tuning can be a difficult process, particularly for more complex pipelines, and the optimized models may be hard to understand. In combination with abstract caches, symbolic pipeline analysis only unfolds its full power if one accepts a loss of precision. The reason for this is that we have no feasible symbolic representation for cache states and we also cannot afford to enumerate the pipeline states when interfacing abstract caches. However, symbolic pipeline analysis is designed to deal with cases of increased complexity because of state explosion. Losing information in order to gain computability is a broadly accepted approach in program analysis. Getting a less precise result, i.e., a coarser WCET bound, is better than getting no result at all.

# CHAPTER 7

# Practical Evaluation

The goal of the symbolic implementation of pipeline domains is to improve the efficiency of pipeline analysis when dealing with many potential pipeline states for the same execution cycle. We discussed in Section 5.3 why such situations arise and why they cannot be handled efficiently by the legacy explicit-state approach. The downside of using a symbolic method for pipeline analysis is the increased sensitivity with respect to the size of the pipeline model. Furthermore, a naïve approach that encodes the complete program in a monolithic symbolic representation is even sensitive to the size of the program. It is therefore mandatory to show that the proposed program decomposition (see Section 6.4.4) is effective, and that the gain in efficiency with respect to the number of explored states outweighs the additional costs (for computing the mappings and for additional BDD operations like reorderings) in practically relevant scenarios.

This chapter describes an implementation of a symbolic pipeline analysis framework that is integrated into the WCET toolchain presented in Section 5.1. The symbolic framework replaces the microarchitectural analysis phase (see Section 5.1.3) of the toolchain. All other phases have been left unchanged. The symbolic pipeline analysis framework is instantiated by the model of the Infineon TriCore 1 [AG08], a processor which is used in many embedded automotive systems. We explain some of the issues that we had to solve to obtain a working prototype.

Finally, we describe a number of experiments that we performed with the prototype implementation. We compare its performance with an equivalent explicit-state implementation. The results show the efficiency of the symbolic implementation and demonstrate the effectiveness of our optimizations. However, they also show that the legacy explicit-state approach remains superior in certain situations. A comprehensive discussion of the results is found at the end of this chapter.

## 7.1. A Framework for Symbolic Pipeline Analyses

The WCET analysis framework described in Section 5.1 is implemented by the commercial WCET tool `aiT` [Abs00]. We chose `aiT` as a basis for integration of the symbolic pipeline analysis framework since it is a mature, commercial tool and we have full access to its source code. As a consequence of this decision, the following design criteria have to be met:

1. The pipeline analyzer must use the same program representation as the value and control flow analyzers. The program representation that is used by the analyses within `aiT` is that of an extended supergraph as introduced in Section 3.3. The supergraph is annotated with additional attributes. The pipeline analysis must also consider the same analysis contexts as the value and control flow analyzers to import their results with maximum precision. In order to guarantee this behavior, we use the exact same supergraph iterator which we generate using the program analysis generator PAG [Mar98].

2. Not only must relevant program information and results from other analyses in the framework be extracted from the extended supergraph. Abstract pipeline states also have to be mapped to program points in the graph and individual instructions must be identified unambiguously by the pipeline model. In both cases, additional context information must be considered.

3. The results of the analysis must be written as context-sensitive basic block attributes in the extended supergraph where they can be accessed by the subsequent path analysis.

Further implementation constraints are imposed by the chosen BDD representation and image computation engine. After evaluating several BDD libraries and model checkers, we decided to take the relevant parts from the VIS system for *Verification Integrated with Synthesis*, which also comprises a symbolic model checker [BHSV$^+$96]. This decision was motivated by the fact that VIS features an advanced image operator [RAB$^+$95] and provides a very convenient interface for building the transition relations of models from Verilog specifications [Che94]. Also, its complete source code is available under a license that does not conflict with the integration into the commercial `aiT` tool. The VIS model checker can be compiled using different BDD libraries. We use the CUDD library [Som09].
The design of the symbolic pipeline analysis framework is depicted in Figure 7.1. The framework reads the following inputs:

- a program for analysis, given as an extended, attributed supergraph,
- a Verilog [TM96] specification of the abstract pipeline model,
- a specification of the model-to-program interface that defines the relations between the model's program dependent variables and the corresponding supergraph attributes.

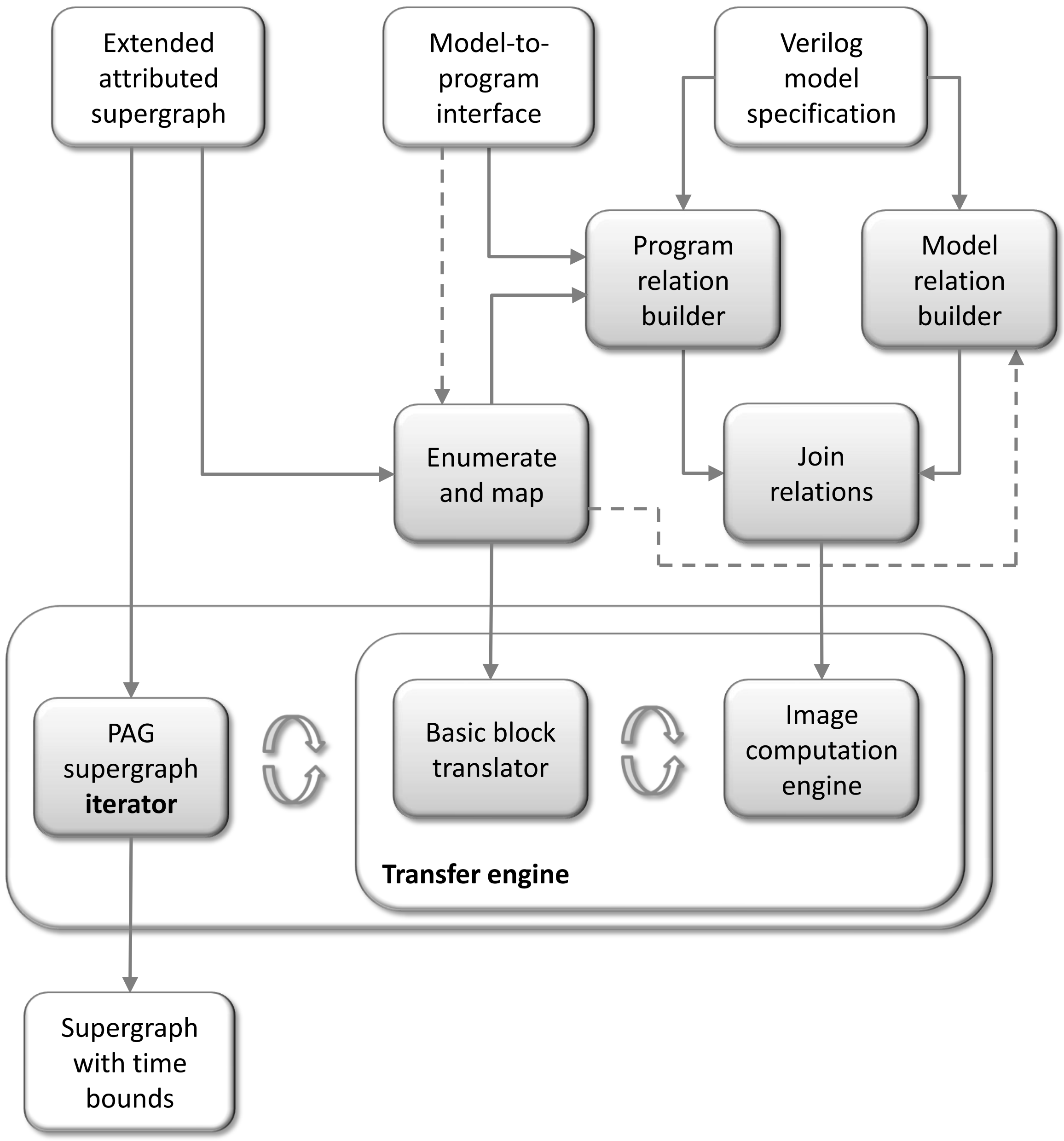


**Figure 7.1.:** Overview of the symbolic pipeline analysis framework. The arrows show the flow of information between the different interacting modules. Dashed arrows indicate that only a very small amount of information is transferred whereas solid arrows indicate major inputs and outputs. The turning arrows denote iterations over the control flow graph and the model state space, respectively. The inputs for the framework are depicted in the three top-level boxes. The produced result is represented by the bottom-level box.

A setup stage computes the symbolic transition relations of the model and the program. It begins with the *enumerate and map* module which reads the supergraph and enumerates all instructions in all analysis contexts. The computed mappings determine the number of model variables that are required for enumerating all instructions in all contexts. The *model relation builder* then specializes the Verilog model to use the smallest possible number of model variables for the analyzed program. The specialized Verilog model is compiled into the netlist format BLIF [BCH+91] which is then read by the VIS model checker to produce the symbolic transition relation of the model. The symbolic program transition relation is generated by the *program relation builder* based on the Verilog model specification and the model-to-program interface. Instruction attributes are accessed through the mappings that have been computed by the *enumerate and map* module. The last step of the setup stage is the *join relations* module which constructs the combined model and program relation that is required for the state traversal.

The analysis engine is divided into two parts. The PAG supergraph iterator performs the fixed point iteration on the interprocedural control flow graph. Each time the iterator reaches a basic block in an analysis context with a modified set of incoming pipeline states, it calls the symbolic transfer engine for each outgoing edge of the basic block. The transfer engine implements the algorithm of Figure 6.6 (or Figure 6.9 if the optional decomposition is enabled) which traverses the reachable pipeline states within the scope of the analyzed basic block. The number of state transitions, i.e., the computed number of execution cycles, is annotated at the extended supergraph and the outgoing set of states is given back to the supergraph iterator. The result of the analysis is an extended supergraph annotated with context-sensitive time bounds for all reachable basic blocks.

The following sections describe important modules and aspects of the implementation in more detail.

## 7.1.1. Enumerate and Map

The first operation of the framework is to read the supergraph and to enumerate all instructions in all analysis contexts. The result of this operation is a bijective mapping between a set of natural numbers and tuples of addresses and contexts. This mapping is the basis for the address and context compression as described in Section 6.4.2. As stated there, the numbering must preserve the order of instruction addresses to allow for address arithmetic in the abstract pipeline model. We ensure this in the collection phase that collects all tuples of addresses and contexts in the analyzed program. The collection phase is implemented by a single-pass recursive algorithm over the supergraph, starting from the analysis entry point.

The collection algorithm (Algorithm 7.1) is called with a collection list $lst$, a basic block $b$, and a context number $ctx$. For the initial call, the $lst$ argument is empty, $b$ is the start block of the supergraph, and $ctx$ is the default context number 0. In line 1 the algorithm first checks whether the current basic block has already been reached in the current context. If not, the current block and context tuple is tagged as being reached in line 2. For each instruction in the current block, the loop in lines 3-5 appends a tuple of the instruction address and the current context to the collection list *lst*. The second loop in lines 6-11 then collects the tuples from lexical successor blocks. To this end, the expression $first.addr = last.addr + last.width$ in line 9 checks whether the first instruction of the successor block $b'$ is a lexical successor of the last instruction in *b*. Finally, the loop in lines 12-17 collects the tuples from the remaining successor blocks. When the algorithm terminates, it returns all reachable instruction and context tuples of the program in lexical order of their addresses. Note that the algorithm in Algorithm 7.1 is much simplified for illustrative purposes. Traversing the supergraph also involves dealing with special nodes (particularly call and return nodes) and edges (e.g., local edges). Also the check in line 9 of the algorithm is more difficult for the same reason.

**Algorithm 7.1**: `collect` ( $lst, b, ctx$ )

```
1  if not reached($b, ctx$) then
2      tag($b, ctx$);
3      forall $l \in b$ do
4          $tuple \leftarrow (l.addr, ctx)$;
5          $lst \leftarrow$ append($lst, tuple$);
6      forall $(b', ctx') \in$ succs($b, ctx$) do
7          $first \leftarrow$ first($b'$);
8          $last \leftarrow$ last($b$);
9          if $first.addr = last.addr + last.width$ then
10             $lst' \leftarrow$ collect($lst, b', ctx'$);
11             $lst \leftarrow$ join($lst, lst'$);
12     forall $(b', ctx') \in$ succs($b, ctx$) do
13         $first \leftarrow$ first($b'$);
14         $last \leftarrow$ last($b$);
15         if $first.addr \neq last.addr + last.width$ then
16             $lst' \leftarrow$ collect($lst, b', ctx'$);
17             $lst \leftarrow$ join($lst, lst'$);
18 return $lst$;
```

The next phase of the module traverses the collected table and inserts padding tuples. Padding is required at the boundaries of addressing holes and at the end of the program. Although these addresses can never be executed by a correct run of the program, they may be loaded into the pipeline by prefetching or speculation. The number of required padding tuples at an address hole is bounded by $\frac{\bar{c}}{2}$, where $\bar{c}$ is the overlap bound of the considered pipeline model. Further padding is required if the pipeline model features different instruction sizes. Such pipeline models may compute addresses that do not correspond to instruction boundaries.

The resulting table of address and context tuples is then enumerated to obtain a *global mapping* between natural numbers and locations in the supergraph. A set of *local mappings* for each basic block in each analysis context is computed by the final, optional phase of the module. The local mappings are required for implementing the program decomposition of Section 6.4.4. To compute the local mapping of a basic block in a specific context, we determine the corresponding enumeration window on the supergraph as depicted in Figure 6.8 using the *preds* and *succs* functions of Equation 6.6 and Equation 6.7, respectively. We then enumerate all instructions within the window and store a mapping from the local to the global enumeration. This mapping is used by the basic block translator in the analysis engine to translate the mappings between neighboring basic blocks. Since each local mapping is related to the unambiguous global mapping, the local mappings of different basic blocks can also be unambiguously translated.

The outputs of the enumerate and map module are the global and local mappings between natural numbers and tuples of instructions and analysis contexts. The optional computation of local mappings is controlled by a switch at analysis time.

## 7.1.2. The Model Relation Builder

The enumerate and map module produces mappings between natural numbers and tuples of instructions and analysis contexts. If the optional program decomposition optimization (see Section 6.4.4) is disabled, it only computes a global mapping for implementing address compression (see Section 6.4.2). Otherwise, it also computes a set of local mappings which represent the enumeration windows for program decomposition as depicted in Figure 6.8. The number of tuples in the global mapping *or* the number of tuples in the largest local mapping (if local mappings have been computed) is passed to the model relation builder. The model relation builder then instantiates the Verilog model specification such that each instruction address in the model is represented by a number of state variables logarithmic in the provided bound. Hence, it constructs a smaller model if the program representation is more compact. If local mappings have been computed, the model size does not depend on the program size anymore as discussed in Section 6.4.4.

The instantiated Verilog specification of the pipeline model is compiled into the BLIF format using the Verilog to BLIF compiler `vl2mv` [Che94] that is part of VIS. The BLIF file contains a netlist representation of the model as depicted in the example of Figure 6.2. The module then invokes the BLIF parser of VIS which reads the netlist and constructs a list of functions, each of which represents the transition relation of a single latch. For each function, we then create its symbolic transition relation as explained in Section 6.2.3. Finally, the symbolic transition relations are combined into the partitioned transition relation of the model (see Section 6.2.1 and Section 6.4.1). We use the proven default heuristics of VIS to determine an efficient partitioning. The partitioning strategy is described in [RAB$^+$95].

### 7.1.3. The Program Relation Builder

The inputs for the program relation builder are the model-to-program interface and the mapping tables computed by the enumerate and map module. The model-to-program interface specifies which variables of the Verilog model are program dependent and under which conditions they are updated. The interface description further specifies how the required input information can be retrieved from attributes of the supergraph. The program relation builder performs one pass over all instructions and contexts and generates transition relations that restrict the next states depending on the values of relevant instruction attributes. These relations are then conjoined into the partitioned program relation. The partitioning is controlled by a fixed BDD size limit. If the current partition exceeds the threshold, it is closed and further relations are added to the next partition. The program relation builder also takes care of model specific optimizations that require information about the analyzed program. For example, for the TriCore 1 pipeline model, the static precomputation of pipeline stall conditions is performed by this module. Stall conditions are computed by a data dependency analysis with restricted scope and encoded as pipeline model inputs as described in Section 6.4.3.

### 7.1.4. Iterator, Transfer Function, and Control Flow Handling

The supergraph iterator and the transfer function of the symbolic pipeline analysis framework have been generated using the PAG program analysis generator with an empty analysis specification. The generated transfer function was then modified to implement the algorithms of Figure 6.6 and Figure 6.9. However, the actual implementation deviates slightly from these rather straightforward definitions. In particular, it features more sophisticated solutions for dealing with complex control flow. The algorithms of Figure 6.6 and Figure 6.9 only deal with the basic case of two outgoing edges of a basic block in cases of a conditional branch (true edge, false edge). To this end, they create two BDDs for the two possible control flow successors at the point where the control flow decision is made in the pipeline. The same solution can also be applied to more complex cases where a block has many control flow successors, e.g., in cases of computed branches for implementing switch tables.

The control flow handling is more complicated if the graph contains sequences of very short basic blocks with several outgoing and incoming edges. It can happen that all instructions of several subsequent basic blocks enter the pipeline before the last instruction of the first block retires. We handle these cases using a sufficiently large look-ahead (determined by the overlap bound of the pipeline model, see Theorem 6.1) and decomposing the set of pipeline states into many BDDs which flow to the right successor blocks. The same control flow problem is also encountered in explicit-state analyzers. They store additional history information in the pipeline state to handle such cases. Our solution deliberately avoids storing additional information in the pipeline state in order to keep the pipeline model small.

At call edges we use a simpler solution which filters infeasible incoming states using carefully designed Boolean expressions. The basic idea is that a function can only be entered by a pipeline state which has recently processed a call instruction with this function as its target. This filtering reduces the complexity of the general look-ahead control flow handler. The Boolean expressions for identifying feasible entry states are part of the model-to-program interface.

### 7.1.5. Debugging Interface

The symbolic pipeline analysis framework features a visual debugging interface to help designers debug their pipeline models. The interface offers two modes of operation:

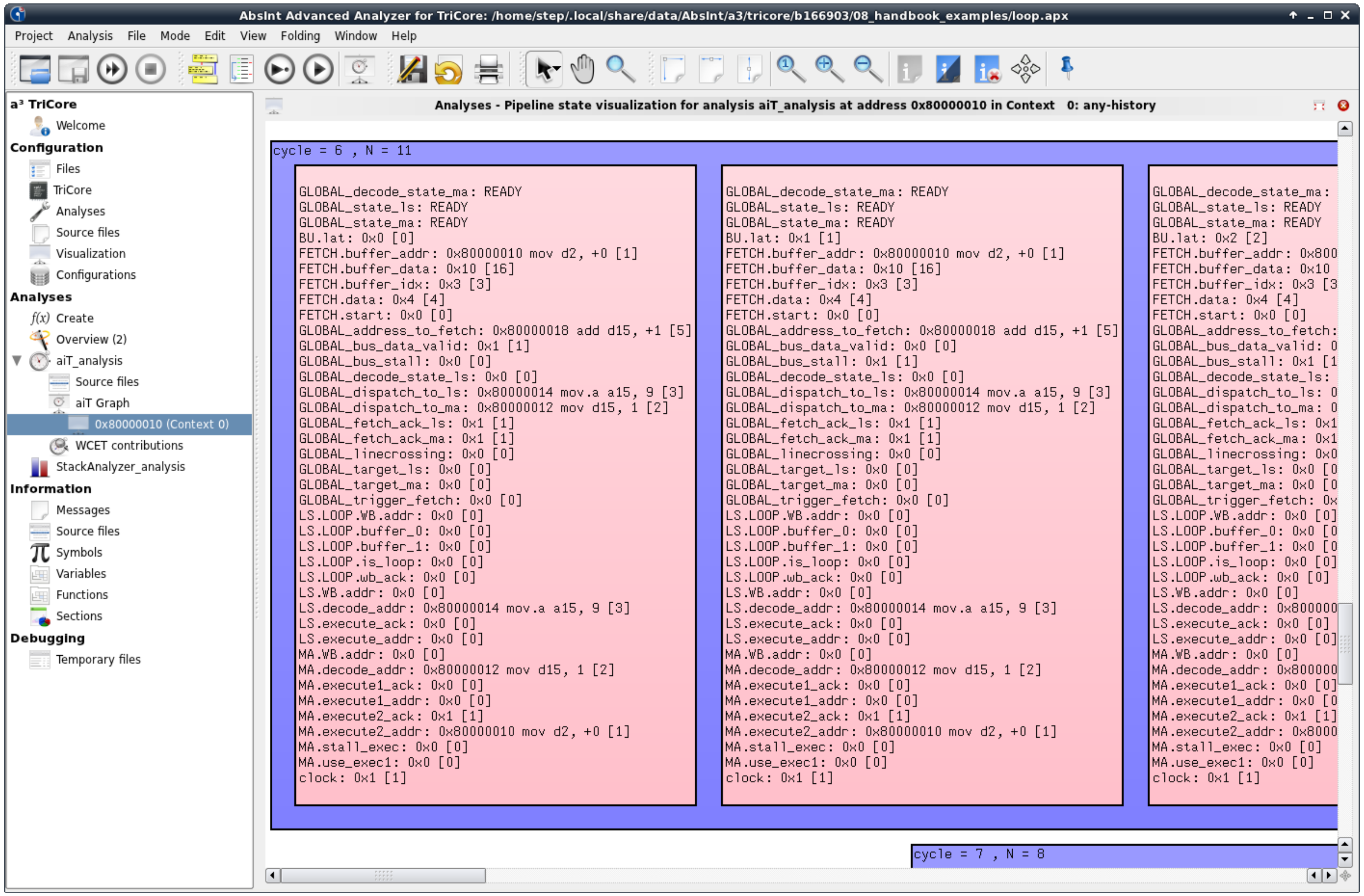
1. In *fixed point mode*, the interface first shows the basic block representation of the analyzed program. For each basic block, the user selects an analysis context for which the abstract basic block trace (see Definition 5.11) of the pipeline is then displayed. This debugging mode is fully integrated with the GUI of the `aiT` WCET tool.
2. In *iteration mode*, one can step through the computations of the fixed point iteration on the supergraph. In each step, the interface shows the last abstract trace for each basic block and context. The mappings between the static contexts at neighboring basic blocks are explicitly displayed. This debugging mode is only available for developers of pipeline analyses. It is not integrated with the GUI of the `aiT` WCET tool.

In both operation modes, the abstract basic block traces are displayed as a trace of sets of abstract pipeline states as in Figure 6.5. All pipeline states that are reached in the same cycle are explicitly listed. To this end, the implementation relies on a VIS function for enumerating the satisfying assignments of the Boolean function that represents the current set of pipeline states. For each element of this enumeration, the model variables are listed with their current value. The compressed instruction and context numbering is translated back into the 32-bit program addresses and the supergraph context numbers using the computed mapping tables. Since enumerating the satisfying assignments is expensive for large sets, the explicit enumeration of pipeline states for debugging is limited to 512 states per cycle.

## 7.2. Experimental Results

The described framework for symbolic pipeline analyses has been instantiated with a pipeline model of the Infineon TriCore 1 microprocessor. The choice of this processor was motivated by the following considerations:

- The pipeline of the Infineon TriCore 1 is sufficiently complex to exhibit timing anomalies, as illustrated by Figure 7.4. Hence, the pipeline analysis requires



```
cycle = 6 , N = 11

GLOBAL_decode_state_ma: READY
GLOBAL_state_ls: READY
GLOBAL_state_ma: READY
BU.lat: 0x0 [0]
FETCH.buffer_addr: 0x80000010 mov d2, +0 [1]
FETCH.buffer_data: 0x10 [16]
FETCH.buffer_idx: 0x3 [3]
FETCH.data: 0x4 [4]
FETCH.start: 0x0 [0]
GLOBAL_address_to_fetch: 0x80000018 add d15, +1 [5]
GLOBAL_bus_data_valid: 0x1 [1]
GLOBAL_bus_stall: 0x0 [0]
GLOBAL_decode_state_ls: 0x0 [0]
GLOBAL_dispatch_to_ls: 0x80000014 mov.a a15, 9 [3]
GLOBAL_dispatch_to_ma: 0x80000012 mov d15, 1 [2]
GLOBAL_fetch_ack_ls: 0x1 [1]
GLOBAL_fetch_ack_ma: 0x1 [1]
GLOBAL_linecrossing: 0x0 [0]
GLOBAL_target_ls: 0x0 [0]
GLOBAL_target_ma: 0x0 [0]
GLOBAL_trigger_fetch: 0x0 [0]
LS.LOOP.WB.addr: 0x0 [0]
LS.LOOP.buffer_0: 0x0 [0]
LS.LOOP.buffer_1: 0x0 [0]
LS.LOOP.is_loop: 0x0 [0]
LS.LOOP.wb_ack: 0x0 [0]
LS.WB.addr: 0x0 [0]
LS.decode_addr: 0x80000014 mov.a a15, 9 [3]
LS.execute_ack: 0x0 [0]
LS.execute_addr: 0x0 [0]
MA.WB.addr: 0x0 [0]
MA.decode_addr: 0x80000012 mov d15, 1 [2]
MA.execute1_ack: 0x0 [0]
MA.execute1_addr: 0x0 [0]
MA.execute2_ack: 0x1 [1]
MA.execute2_addr: 0x80000010 mov d2, +0 [1]
MA.stall_exec: 0x0 [0]
MA.use_exec1: 0x0 [0]
clock: 0x1 [1]
```

```
GLOBAL_decode_state_ma: READY
GLOBAL_state_ls: READY
GLOBAL_state_ma: READY
BU.lat: 0x1 [1]
FETCH.buffer_addr: 0x80000010 mov d2, +0 [1]
FETCH.buffer_data: 0x10 [16]
FETCH.buffer_idx: 0x3 [3]
FETCH.data: 0x4 [4]
FETCH.start: 0x0 [0]
GLOBAL_address_to_fetch: 0x80000018 add d15, +1 [5]
GLOBAL_bus_data_valid: 0x0 [0]
GLOBAL_bus_stall: 0x1 [1]
GLOBAL_decode_state_ls: 0x0 [0]
GLOBAL_dispatch_to_ls: 0x80000014 mov.a a15, 9 [3]
GLOBAL_dispatch_to_ma: 0x80000012 mov d15, 1 [2]
GLOBAL_fetch_ack_ls: 0x1 [1]
GLOBAL_fetch_ack_ma: 0x1 [1]
GLOBAL_linecrossing: 0x0 [0]
GLOBAL_target_ls: 0x0 [0]
GLOBAL_target_ma: 0x0 [0]
GLOBAL_trigger_fetch: 0x0 [0]
LS.LOOP.WB.addr: 0x0 [0]
LS.LOOP.buffer_0: 0x0 [0]
LS.LOOP.buffer_1: 0x0 [0]
LS.LOOP.is_loop: 0x0 [0]
LS.LOOP.wb_ack: 0x0 [0]
LS.WB.addr: 0x0 [0]
LS.decode_addr: 0x80000014 mov.a a15, 9 [3]
LS.execute_ack: 0x0 [0]
LS.execute_addr: 0x0 [0]
MA.WB.addr: 0x0 [0]
MA.decode_addr: 0x80000012 mov d15, 1 [2]
MA.execute1_ack: 0x0 [0]
MA.execute1_addr: 0x0 [0]
MA.execute2_ack: 0x1 [1]
MA.execute2_addr: 0x80000010 mov d2, +0 [1]
MA.stall_exec: 0x0 [0]
MA.use_exec1: 0x0 [0]
clock: 0x1 [1]
```

```
GLOBAL_decode_state_ma:
GLOBAL_state_ls: READY
GLOBAL_state_ma: READY
BU.lat: 0x2 [2]
FETCH.buffer_addr: 0x800
FETCH.buffer_data: 0x10
FETCH.buffer_idx: 0x3 [3
FETCH.data: 0x4 [4]
FETCH.start: 0x0 [0]
GLOBAL_address_to_fetch:
GLOBAL_bus_data_valid: 0
GLOBAL_bus_stall: 0x1 [1
GLOBAL_decode_state_ls:
GLOBAL_dispatch_to_ls: 0
GLOBAL_dispatch_to_ma: 0
GLOBAL_fetch_ack_ls: 0x1
GLOBAL_fetch_ack_ma: 0x1
GLOBAL_linecrossing: 0x0
GLOBAL_target_ls: 0x0 [0
GLOBAL_target_ma: 0x0 [0
GLOBAL_trigger_fetch: 0x
LS.LOOP.WB.addr: 0x0 [0]
LS.LOOP.buffer_0: 0x0 [0
LS.LOOP.buffer_1: 0x0 [0
LS.LOOP.is_loop: 0x0 [0]
LS.LOOP.wb_ack: 0x0 [0]
LS.WB.addr: 0x0 [0]
LS.decode_addr: 0x800000
LS.execute_ack: 0x0 [0]
LS.execute_addr: 0x0 [0]
MA.WB.addr: 0x0 [0]
MA.decode_addr: 0x800000
MA.execute1_ack: 0x0 [0]
MA.execute1_addr: 0x0 [0
MA.execute2_ack: 0x1 [1]
MA.execute2_addr: 0x8000
MA.stall_exec: 0x0 [0]
MA.use_exec1: 0x0 [0]
clock: 0x1 [1]
```

```
cycle = 7 , N = 8
```

Figure 7.2.: Screenshot of the debugging interface that is integrated into the GUI of the aiT WCET tool. It shows the first 2 of 11 pipeline states in the 6th execution cycle of the basic block at address 0x80000010 in its first analysis context.

an exhaustive state traversal and may therefore run into problems of state explosion. It is possible to setup experiments where the analysis reaches a substantial subset of the pipeline's state space.

- The `aiT` WCET analyzer already features an explicit-state model of the TriCore 1. The implementation consists of the pipeline core model and modules that represent the relevant external hardware of several TriCore 1 variants, e.g., the TriCore 1766, the TriCore 1796, and the TriCore 1797. This modular design allows the exclusive analysis of the pipeline core. Thus, we can study the pipeline analysis alone without having to consider the interaction with caches and flash buffers.

The following subsections describe the design of the TriCore 1, the characteristics of its abstract pipeline model, and experiments to compare the performance of a symbolic versus an explicit-state implementation of pipeline analysis for TriCore 1.

### 7.2.1. The Infineon TriCore 1

The Infineon TriCore 1 [AG08, Zar01] is a microprocessor architecture for embedded systems. It is used in real-time systems in the automotive industry, e.g., for engine control. The design of the processor core is depicted in Figure 7.3. It features two major pipelines and one minor pipeline:

**The integer pipeline (INT)** handles data arithmetic and conditional jumps. The pipeline features a decode, execute, and writeback stage. Multiply-accumulate instructions are handled by a dedicated execution path with two stages.

**The load/store pipeline (LS)** handles loads/stores, address arithmetic, unconditional jumps and calls. It features a decode, execute, and writeback stage. Zero-overhead loop instructions are passed from the decode unit into the dedicated loop pipeline.

**The loop pipeline (L)** is reserved for zero-overhead loop instructions. The pipeline has no decode stage but receives the address and target of a zero-overhead loop from the decode stage of the LS pipeline. This information is stored in the loop cache buffer until the execution of the loop has finished or the information is replaced by another zero-overhead loop.

All pipelines share a common fetch unit which reduces instruction fetch latencies using a 16 byte prefetch buffer. TriCore instructions can be either 2 or 4 bytes wide. Hence, the prefetch buffer can hold up to 8 instructions. The Infineon TriCore can issue one instruction per cycle into each of the major pipelines. Hence, at most two instructions per cycle can be issued. The processor thereby proceeds in strict program order and uses a set of static rules to decide whether two instructions can be issued in parallel. Based on these rules, compilers may optimize execution speed by computing an instruction schedule that provides a high throughput. For improved performance, the TriCore 1 architecture also features static branch prediction.

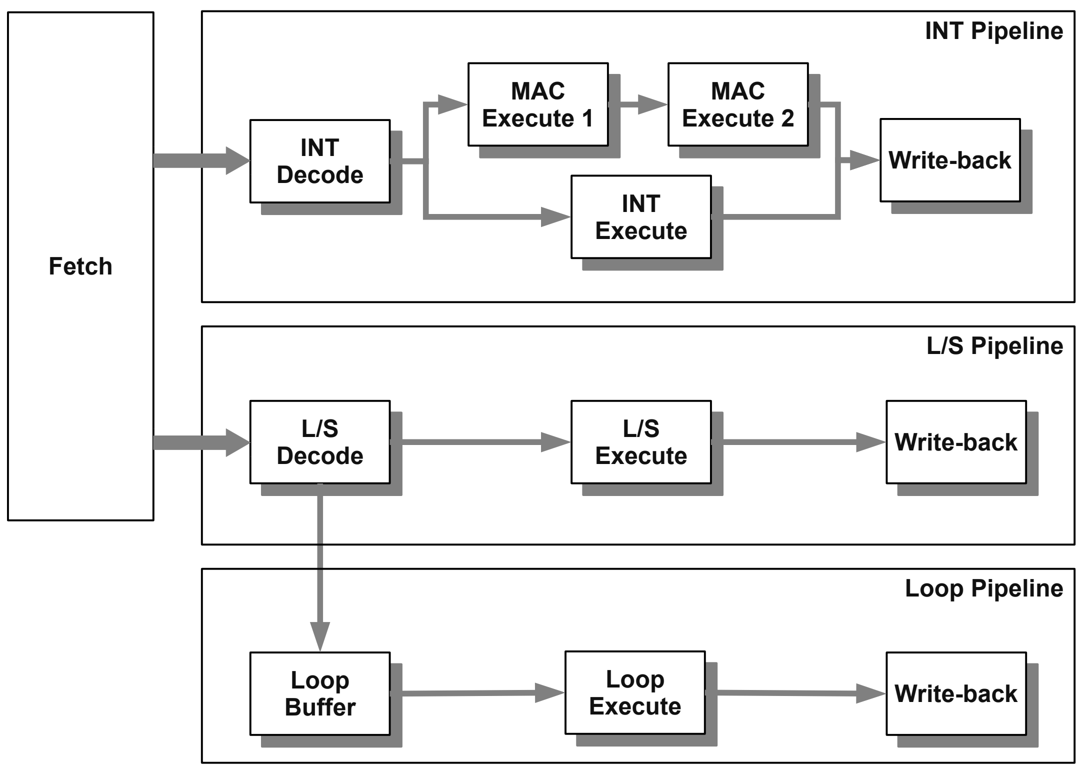


**Figure 7.3.:** Structure of the TriCore 1 pipeline as given in [Zar01].

### 7.2.2. A Timing Anomaly Example

The TriCore 1 has been designed with predictability in mind. However, Figure 7.4 shows that already the presence of the prefetch buffer suffices to introduce a timing anomaly if the latency of an instruction fetch is not precisely known at analysis time. The figure shows two cases of instructions flowing through the prefetch buffer into the two parallel pipelines LS and INT. In cycle 0 the first four instructions have just been fetched. While the next instruction fetch of E and F is running, the processing of instruction A stalls in the decode stage because the INT pipeline is still occupied, e.g., by a long running division operation. Several implementations of the TriCore 1 feature additional flash memory. Depending on the chosen implementation and its configuration an instruction fetch can have a basic latency of four cycles. If the state of the flash memory is not precisely known at the time of an instruction fetch, the pipeline analysis must consider a possible additional delay of one cycle. Figure 7.4 shows the two cases. The upper part depicts the local worst-case, i.e., the state of the flash adds a delay of one cycle to the basic latency of four cycles. The instructions B and C proceed into the pipeline just before the fetched instructions E and F arrive. Both instructions can be stored in the prefetch buffer and issued in parallel in cycle 7. The whole sequence has been issued into the pipelines after 8 cycles.

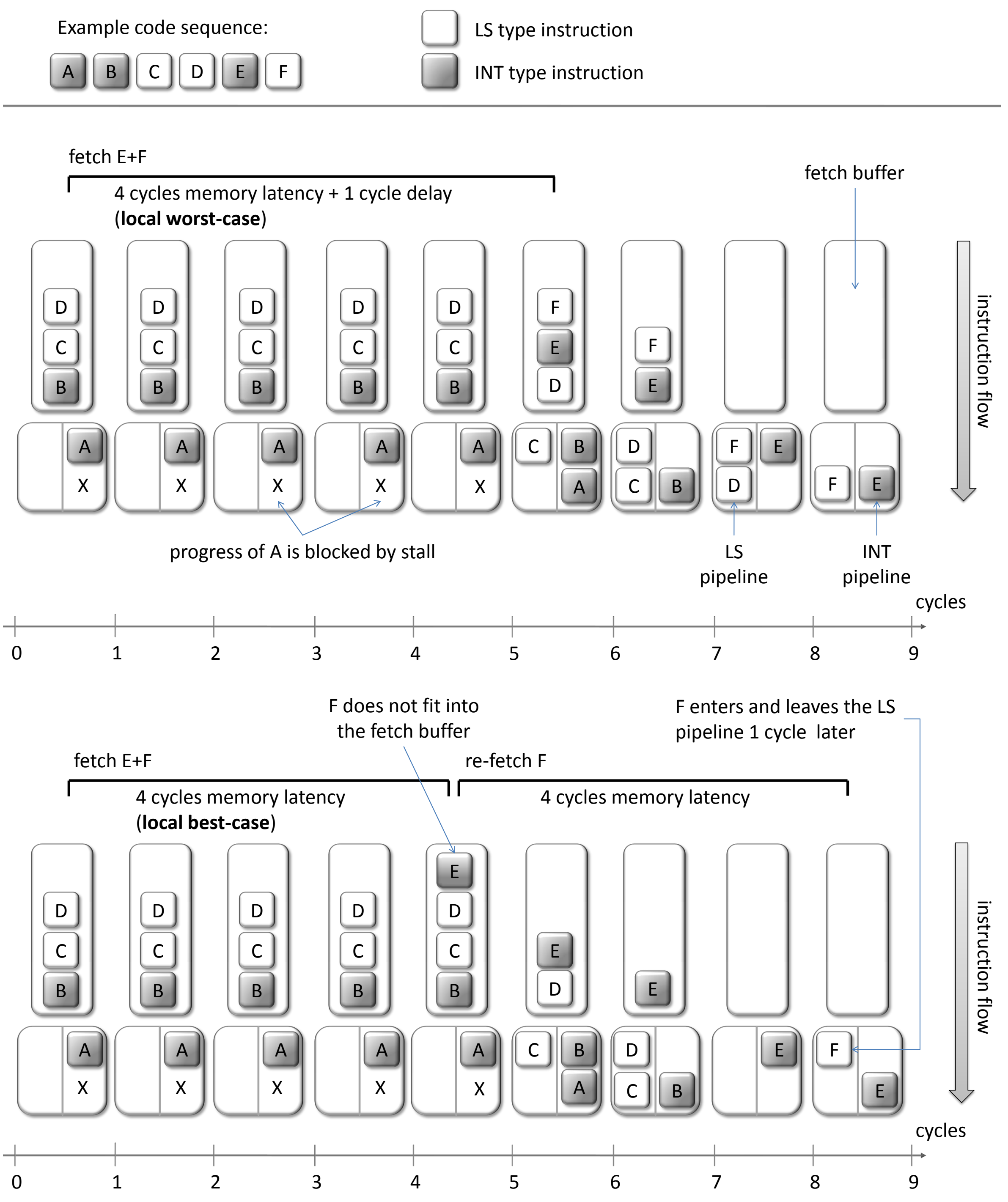


**Figure 7.4.:** Timing anomaly example for TriCore 1. The figure shows the evolution of the fetch buffer and the first two stages of the LS and INT pipelines in a situation were the memory access latency may differ by one cycle depending on the state of a flash memory controller. All instructions of the example code sequence are 4 bytes wide.

In the local best-case situation depicted in the lower part of the figure, the running instruction fetch returns already after four cycles. At that time instructions B and C have not yet proceeded into the pipelines. Therefore only instruction E can be filled into the prefetch buffer; instruction F must be fetched again. Because of the basic delay of four cycles per instruction fetch, instruction F only arrives in cycle 9. Hence it takes 9 cycles until the whole sequence has been issued into the pipelines. If all instructions in the sequence flow smoothly through the pipelines, the overall execution time of both cases also differs by one cycle. Hence, the flash memory state that causes a delay of one cycle leads to a lower execution time on the whole code sequence.

### 7.2.3. TriCore 1 Pipeline Models

A commercial TriCore 1 model including all relevant peripheral hardware such as buses, flash modules, buffers and caches is available in `aiT`. Its representation requires 500 bytes per abstract pipeline state plus 196 bytes for the state of the peripheral components (not counting the dynamically sized abstract cache). Note that explicit-state pipeline models represent instructions and contexts as pairs of 32 bit pointers/integers. The space required to store these pairs dominates the size of abstract pipeline states.[1] Analysis of the TriCore 1 model exhibits two sources of non-determinism:

1. imprecise information about data memory accesses (intervals instead of precise addresses).

2. imprecise information about the state of peripheral components that influence the timing of data memory accesses and instruction fetches.

A single unclassified memory access in this model causes the analysis to explore up to 64 next states per present state. A series of unclassified memory accesses may quickly multiply this number. Imprecise information about the state of caches, flash modules and buffers may introduce further non-determinism.
Our symbolic TriCore 1 model has been modeled after the existing explicit-state model. It conservatively approximates the timing of the pipeline core. The explicit-state model has been validated by comparison with hardware traces of programs running from the on-chip scratch pad memory. The symbolic implementation has been validated by comparison with analysis traces of the explicit-state analysis. Using the presented compression techniques, we arrive at a very compact symbolic representation. Its size ranges from 163 to 333 state bits, depending on the configuration and on the analyzed program.

[1]The memory requirements of the commercial model could be reduced without using BDDs. However, this would increase computation time which is the main bottleneck for explicit-state pipeline analyses.

### 7.2.4. Performance Comparison

Without caches and flash buffers, the symbolic pipeline model for TriCore 1 is deterministic since the processor core can be modeled accurately. To study the effects of state explosion, we introduce non-determinism by intentionally losing precision on the latencies of instruction fetches. This forces the analysis to explore the different possible prefetch buffer states and interleavings of instructions in the pipeline. The resulting behavior of the analysis corresponds closely to the behavior of the full model, including caches and flash buffers, in cases of unknown data memory accesses or cache misses. The commercial TriCore 1 model has been modified accordingly, i.e. the behavior of both models is identical. This setup allows us to study state explosion of pipeline states in isolation, without dealing with the additional complexity of cache and bus models. Further, it gives us control over the number of states that are explored by the analysis. We use it to investigate how both implementations scale up in cases of state explosion. We compare the performance of the explicit- and symbolic-state implementations of the TriCore 1 pipeline core by analyzing the following programs:

**Dhry 2.1** is the Dhrystone [Wei88] integer CPU performance benchmark.

**EDN** is a benchmarks suite which comprises several DSP algorithms [EDN88] like filters, matrix multiplication and FFT.

**EC** is a closed source industrial automotive software for engine control, one of the main application areas for TriCore 1 CPUs. We analyze its 3 major periodic tasks which we call task A, B, and C, respectively.

Table 7.1 lists size information for all of the described programs. The second column gives the number of instructions in all analysis contexts, i.e. after virtual inlining and loop unrolling and after eliminating infeasible paths. The EDN benchmark is analyzed with full virtual inlining and unrolling, hence the large number of instructions. The third column lists the number of bits that are required for the global enumeration of all instructions including padding addresses. The last column gives the same information for the optimized implementation of Section 6.4.4. It determines the number of state variables that are required for implementing a single instruction reference in the pipeline model.
Table 7.1 shows that the program decomposition significantly reduces the number of required state variables. For example, in the case of the EDN benchmark we save 9 variables per instruction reference; for the whole TriCore 1 model this optimization saves 135 variables. For other programs, the savings are less significant, e.g., for the analysis of EC task C the decomposition saves only 3 state variables per instruction reference. As expected, the savings are mostly proportional to the size of the analyzed program. However, the comparison of the numbers for Dhry 2.1 with EC task A show that other contributing factors exist[2] but are much less significant.

[2]The control structure of the software also influences the decomposition. In particular computed branches with many successor blocks lead to larger enumeration windows.

| name | # instructions | # bits global | # bits local |
|---|---|---|---|
| Dhry 2.1 | 3361 | 12 | 7 |
| EDN | 69462 | 17 | 8 |
| EC task A | 3214 | 12 | 8 |
| EC task B | 28606 | 15 | 9 |
| EC task C | 1035 | 11 | 8 |

**Table 7.1.:** Program sizes.

### Configurations and Setup

We compare the analysis run-times for different implementations of the pipeline analysis for TriCore 1. The numbers are obtained from executions on an Intel Core 2 Duo processor running at a clock rate of 2.66 GHz. The computer is equipped with 8 GBytes of RAM. The compared implementations are:

**explicit** is the commercial explicit-state implementation [Abs00].

**symbolic** is the basic symbolic implementation (Chapter 6) which relies on a global enumeration of the instructions and uses the initial static variable order of VIS [BHSV$^+$96].

**symbolic dyn** is the same basic symbolic implementation but with automatic dynamic variable reordering by converging window permutation [FMK91, ISY91].

**symbolic+** is the optimized symbolic implementation that decomposes the program (Section 6.4.4) and operates on a smaller, local enumeration of only relevant instructions; this implementation also uses the initial static variable order of VIS.

**symbolic+ dyn** is the same optimized symbolic implementation but with automatic dynamic variable reordering by converging window permutation.

All programs – the two benchmarks and the three tasks of the application – are analyzed with an increasing number of possible instruction fetch latencies starting from 7 and ranging up to 127. Thus, each instruction fetch multiplies the number of states by 7, 15, 31, 63, or 127, respectively. This setup leads to the following effects in the pipeline analysis:

1. For each instruction fetch, the analysis must consider between 7 and 127 possible successor states which initially only differ in the fetch latency. During the evolution of the pipeline this initial difference in latency leads to different interleavings of the instructions in the pipeline. The average[3] number of

[3]The precise number of states depends on the behavior of the pipeline and on the control structure of the program.

concurrently analyzed states grows exponentially from the first to the last experiment.

2. The summarized maximum latencies on every path through the program grow approximately[4] by a factor of 2 between two successive experiments.

Both effects increase the complexity of the analysis. The increase in the number of states means that the analysis must handle more states per cycle update. The increase in latency means that the analysis must compute more cycle updates. Introducing non-determinism through imprecise information about the instruction fetch latency rather than the latency of data memory accesses ensures that we get a more uniform increase in the number of states because instruction fetches are frequent and evenly distributed.
The reported analysis times are given in seconds (except if indicated otherwise) and include all setup costs, e.g., for the static precomputation of control decisions (both explicit-state and symbolic methods), the enumeration of instructions on the supergraph (symbolic methods), and the construction of the transition relations (symbolic methods). The explicit-state analysis has the least setup costs whereas the optimized symbolic analysis with decomposition has the highest setup costs. The costs for the dynamic reordering of BDD variables during the analysis are also included in the reported analysis times.

### Analysis of Benchmarks

For the analysis of the Dhrystone benchmark we set a time limit of 5 hours. The explicit-state implementation completes the initial analysis, which covers 7 possible instruction fetch latencies, in only 3 seconds. Obviously, the legacy method for pipeline analysis is very fast and efficient when considering small sets of pipeline states. Also, its setup costs are negligible. All 4 configurations of the symbolic implementation also complete the analysis within the time limit. However, with total execution times between 4 and 13 minutes, none of them comes close to the performance of the legacy method. Even for this rather small analysis the program decomposition already significantly improves the performance despite the higher setup costs. The same holds for the dynamic variable reordering. The best performing symbolic implementation combines both optimizations. As the subsequent analyses increase the average number of concurrently considered pipeline states, the resulting exponential growth in complexity becomes visible in the execution times of the explicit-state analysis. In contrast, the symbolic implementations only show a linear increase in execution time by a factor of 2, which correlates with the growing number of cycle updates. All symbolic implementations reach the break-even point with the explicit-state analysis "between" the two last analyses (split 63 and split 127) and outperform the legacy method in the last analysis. Figure 7.5 visualizes the results in a chart that compares the performance of the explicit-state implementation with the simplest and the most optimized symbolic implementation.

[4]The actual factor is slightly below 2 because the pipelining hides some fetch cycles.

| config | split 7 | split 15 | split 31 | split 63 | split 127 |
|---|---|---|---|---|---|
| explicit | 3 | 19 | 160 | 1286 | 9524 |
| symbolic | 815 | 1379 | 2609 | 5373 | 7748 |
| symbolic dyn | 329 | 580 | 774 | 2256 | 4090 |
| symbolic+ | 455 | 796 | 1466 | 2869 | 4632 |
| symbolic+ dyn | 256 | 522 | 926 | 1849 | 2788 |

**Table 7.2.:** Dhrystone 2.1 benchmark. Times are given in seconds.

The EDN benchmark is a much larger program, at least from the perspective of pipeline analysis; its size is mostly the result of a large number of analysis contexts which multiply the number of actual instructions. The contexts are created by fully unrolling all loops and recursions. Because of the larger program size, we increase the time limit to 48 hours. While the explicit-state method is again very fast in the first analysis (75 seconds), only the most optimized symbolic implementation (with program decomposition and dynamic variable reordering) finishes within the time limit (after 36 minutes). All other implementations succumb to the increased complexity of operations with larger BDDs caused either by a large number of required state variables or an inapt variable ordering. The break-even point with the explicit-state analysis is already crossed in the second to last analysis (split 63). Again, the explicit-state analysis shows an exponential increase in computation time when scaling up the number of states. For the last analysis it even exceeds the ample time limit. The optimized symbolic implementation shows the expected linear growth. Even for the last analysis it finishes in about 5.5 hours.

| config | split 7 | split 15 | split 31 | split 63 | split 127 |
|---|---|---|---|---|---|
| explicit | 75 | 480 | 3666 | 40266 | > 48 h |
| symbolic+ dyn | 2169 | 4033 | 7965 | 15245 | 19561 |

**Table 7.3.:** EDN benchmark. Times are given in seconds except if indicated otherwise.

### Analysis of Engine Control Tasks

In terms of program size, task A of the engine control software is comparable to the Dhrystone benchmark. The generally higher analysis run-times, which are displayed in Table 7.4, are due to the more complex control structure of the code. The analysis requires more loop iterations until the computation converges to a fixed point. Apart from the generally higher analysis times, the obtained results are very similar to those of the Dhrystone benchmark. All symbolic implementations scale much better with the increasing number of pipeline states. However, for the last analysis (split

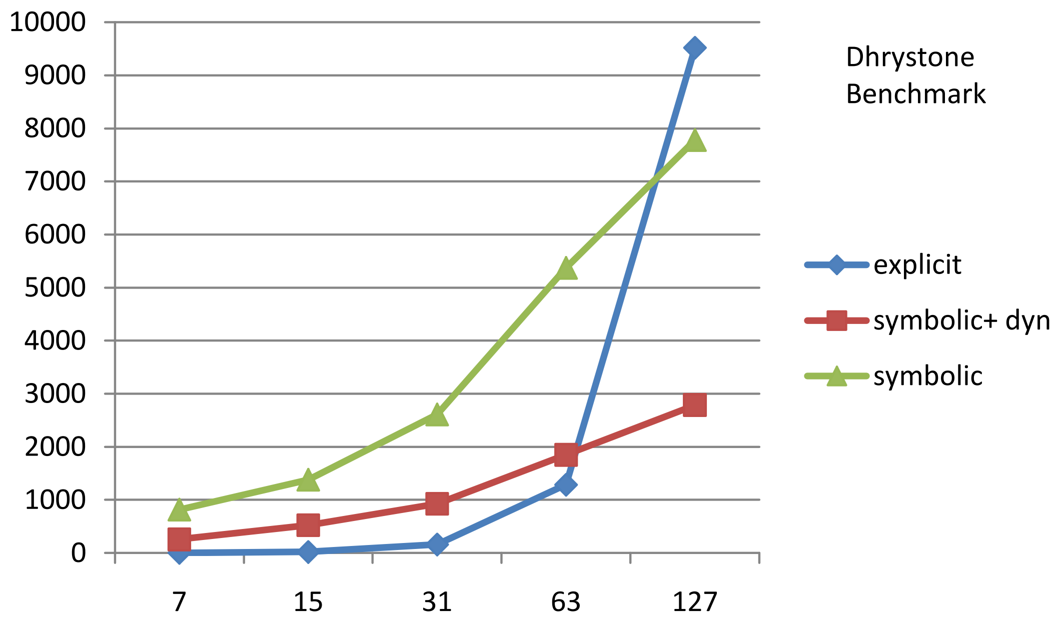


**Figure 7.5.:** Performance comparison chart of the Dhrystone benchmark.

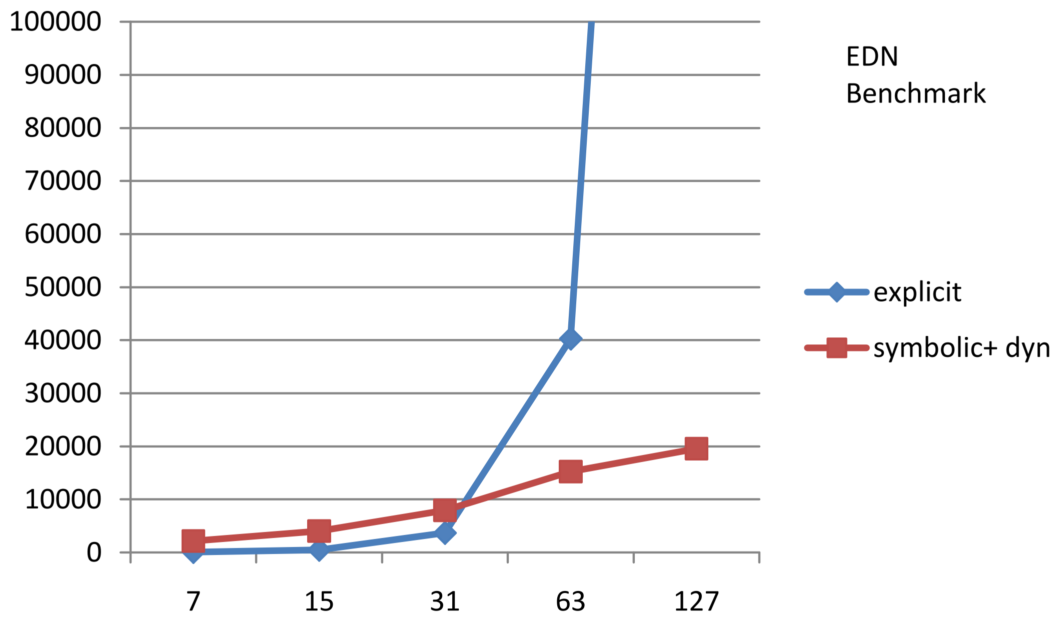


**Figure 7.6.:** Performance comparison chart of the EDN benchmark.

127) only the symbolic implementations that use program decomposition stay within the 5 hours time limit.

| config | split 7 | split 15 | split 31 | split 63 | split 127 |
|---|---|---|---|---|---|
| explicit | 7 | 56 | 432 | 3450 | $> 5$ h |
| symbolic | 2352 | 4706 | 9425 | $> 5$ h | $> 5$ h |
| symbolic dyn | 1131 | 1879 | 2368 | 4502 | $> 5$ h |
| symbolic+ | 1296 | 2480 | 4879 | 9748 | 16069 |
| symbolic+ dyn | 675 | 1174 | 2851 | 3848 | 5905 |

**Table 7.4.:** Engine Control Task A. Times are given in seconds except if indicated otherwise.

Task B of the engine control software is not as big as the EDN benchmark but it is significantly larger than the remaining programs. Its size can only be handled by the explicit-state and the most optimized symbolic implementation. Although the symbolic implementation outperforms the explicit-state implementation in the last analysis, its general performance is worse than for the larger EDN benchmark. The reason for this seemingly surprising result is that the program decomposition optimization is less effective for task B than for EDN. The *bits local* column of Table 7.1 shows that the analysis of EDN needs to enumerate at most $2^8$ instructions per basic block whereas the analysis of task B needs to enumerate up to $2^9$ instructions per basic block. This requires more state variables and leads to a worse performance.

| config | split 7 | split 15 | split 31 | split 63 | split 127 |
|---|---|---|---|---|---|
| explicit | 64 | 459 | 3336 | 26090 | $> 48$ h |
| symbolic+ dyn | 9529 | 14947 | 28141 | 56122 | 101658 |

**Table 7.5.:** Engine Control Task B. Times are given in seconds except if indicated otherwise.

The last of the engine control tasks – task C – is the smallest of the analyzed programs. All implementations finish comfortably within the 5 hours time limit, even for the last configuration (split 127). Again, the best symbolic implementation outperforms the explicit-state implementation in the second to last run. What is interesting about this last experiment is that here the program decomposition does *not* improve the performance of the analysis. It appears that the additional costs for the translation between the basic blocks (see Section 6.4.4) outweigh the saving of 3 state variables per instruction reference (see Table 7.1).

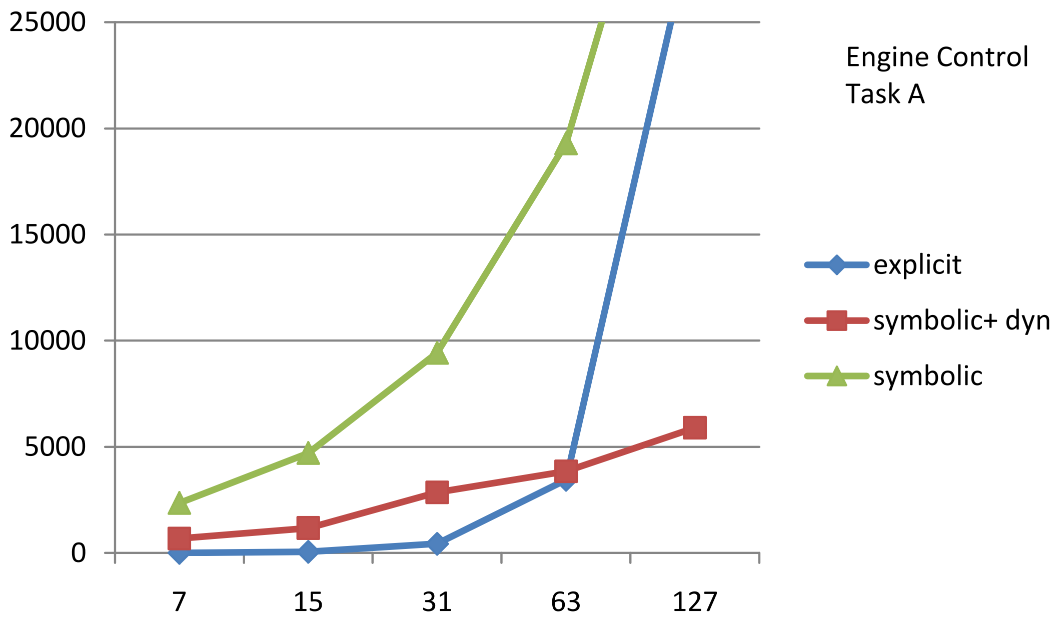


**Figure 7.7.:** Performance comparison chart of engine control task A.

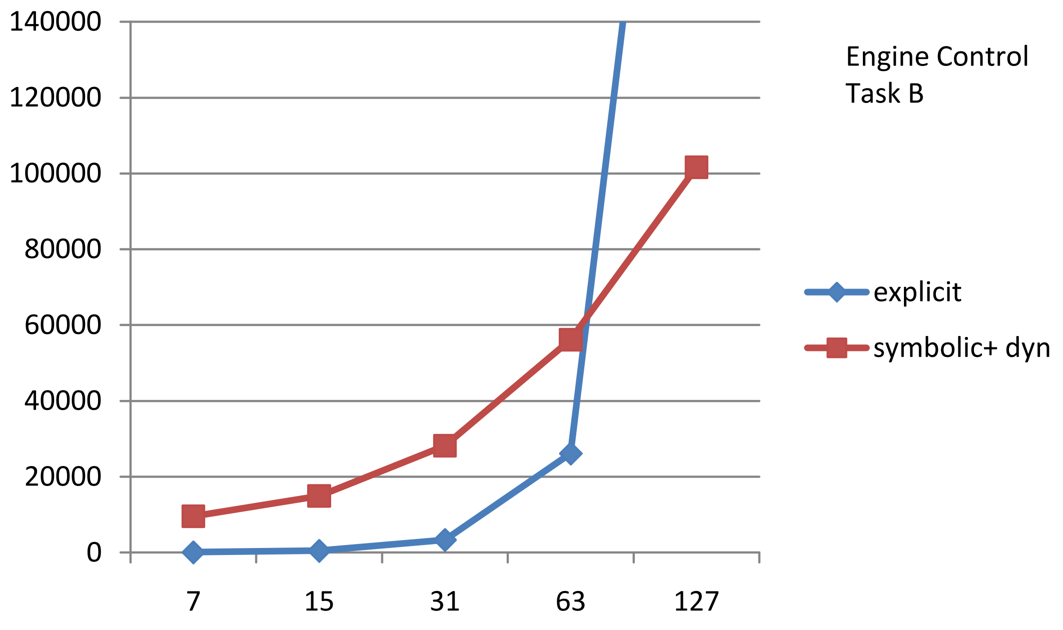


**Figure 7.8.:** Performance comparison chart of engine control task B.

| config | split 7 | split 15 | split 31 | split 63 | split 127 |
|---|---|---|---|---|---|
| explicit | 1 | 9 | 72 | 549 | 4198 |
| symbolic | 230 | 383 | 685 | 1564 | 2034 |
| symbolic dyn | 146 | 179 | 277 | 374 | 692 |
| symbolic+ | 328 | 515 | 1155 | 2367 | 3967 |
| symbolic+ dyn | 227 | 506 | 649 | 1205 | 1860 |

**Table 7.6.:** Engine Control Task C. Times are given in seconds.

### Interpretation of the Results

All experiments show that the symbolic approach for pipeline analysis is indeed less sensitive to the state explosion problem in WCET analysis than the explicit-state approach. If the number of concurrently considered pipeline states grows large enough, the symbolic approach always outperforms the legacy explicit-state implementation. The obtained numbers confirm our expectations. Let us briefly reflect on the relevance of these results with respect to real-life WCET analyses. State explosion problems in WCET analysis are typically associated with unclassified memory accesses. As stated in Section 5.3.3, the commercial TriCore 1 model performs at most 64 *state splits*[5] per unclassified memory access. The break-even point between the best symbolic and the explicit-state implementation in our experiments is around these 64 splits. This indicates that the symbolic method can lead to a better performance in practically relevant scenarios. On the downside the results also show clearly that the basic costs for computing a single cycle update with BDD operations are much higher than the execution of the explicit-state transition function. In particular for small sets of pipeline states, the legacy implementation is always significantly faster. Although it may be possible to reduce the basic costs by additional optimizations, it is unlikely that one can beat the explicit-state implementation on analyses where no state explosion problems occur.

The third observation concerns the program decomposition. Without this optimization none of the two larger programs (EDN and Engine Control Task B) can be analyzed within the time limit. The problem is independent of the number of considered pipeline states. It is caused by the size and complexity of the symbolic transition relations. Enumerating all instructions multiplied with their analysis contexts is infeasible but for the smallest programs. The results show that program decomposition is crucial for the symbolic approach to scale to serious programs.

Finally, the classic problem of BDD variable ordering is also evident in pipeline analysis. The default variable order chosen by VIS is often not good enough for our application. However, the results with dynamic variable ordering are very encourag-

[5]The notion of state split refers to the implementation of the transition function of explicit-state analyzers. In case of a non-deterministic transition the analyzer *splits* each incoming pipeline state into several outgoing states.

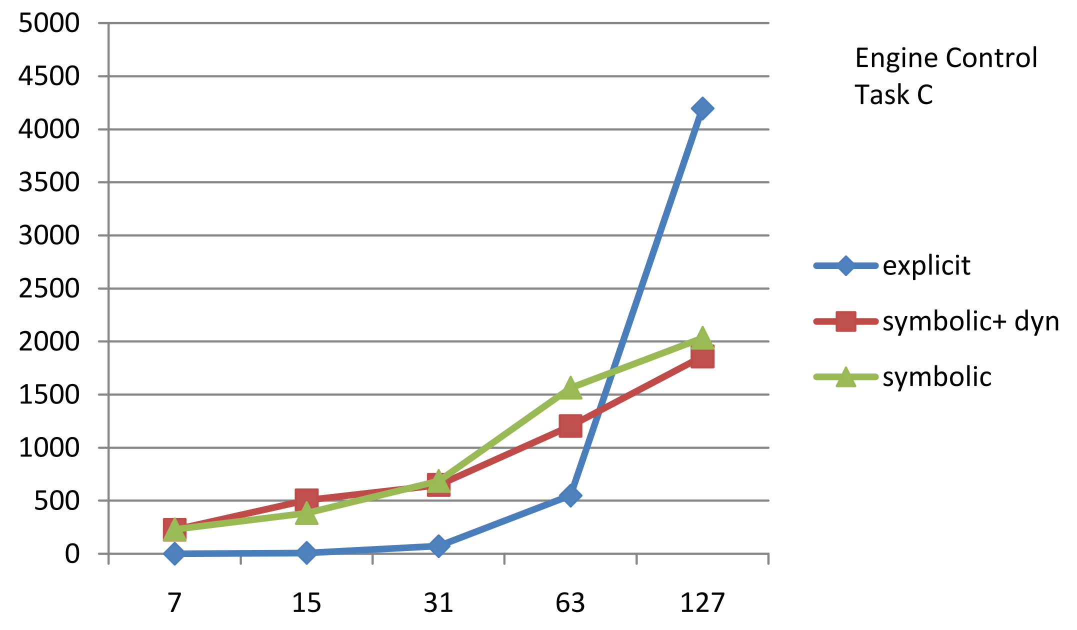


**Figure 7.9.:** Performance comparison chart of engine control task C.

ing. The employed reordering method of window permutation [FMK91, ISY91] was chosen for its speed and consistently good results. However, VIS implements a large number of reordering algorithms that we did not explore exhaustively. It is likely that other advanced reordering algorithms such as sifting [Rud93] will yield similar results. During a single analysis run the dynamic reordering is invoked between 5 and 30 times. All reordering operations are triggered automatically by the BDD library. Hence, the user of a WCET tool that uses our symbolic method for pipeline analysis need not bother about optimizing the variable order.

### Memory Consumption, WCET Bounds, and BDD Sizes

The presented results cover neither the memory consumption of the analysis nor the computed WCET bounds. Information about the WCET has been omitted because it is irrelevant for our purpose. The compared implementations are semantically equivalent, i.e., they agree with respect to the computed WCET bounds. Further, the global WCET bound of a task depends not only on the pipeline analysis but on the interplay of all phases of the `aiT` toolchain. Their precision can only be judged by intimate knowledge of the analyzed software. We lack this knowledge for the engine control application. Information about the precision of `aiT`'s WCET bounds for a processor with caches and complex pipelines has been published in [SLH$^{+}$05].
With respect to the memory consumption, the compared implementations differ significantly. However, our experience with using WCET tools in an industrial context

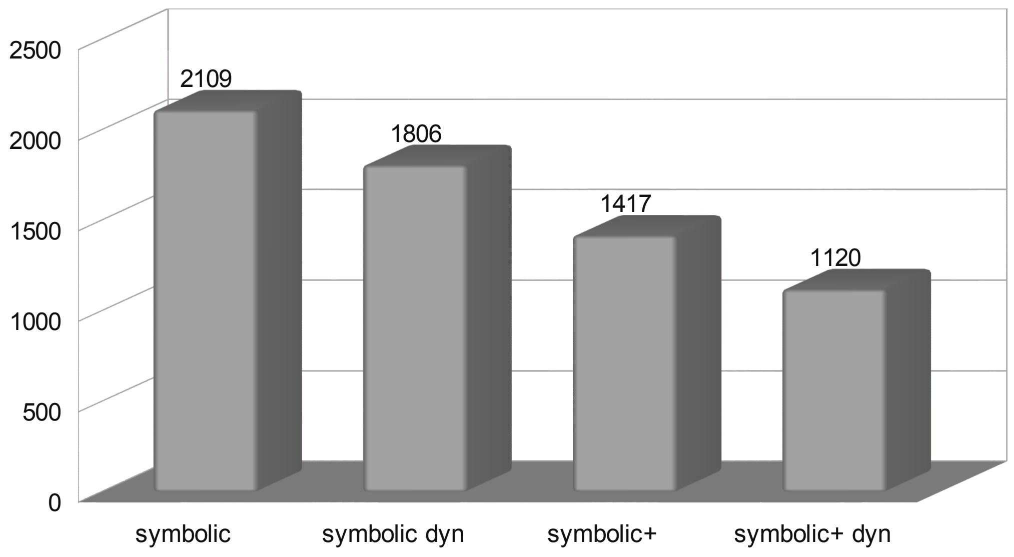


**Figure 7.10.:** Average BDD sizes for Dhrystone with 7 splits per fetch.

indicates that pipeline analysis is much more limited by computation time than by memory constraints. This is confirmed by our experiments. During the experiments none of the analyses reached the 8 GByte limit of the test computer. The observed memory consumption of the explicit-state implementation was always below 4 GByte. For the symbolic implementation we never observed a physical memory consumption of more than 1 GByte. Modern desktop computers are routinely equipped with at least 4 GByte of RAM, hence memory consumption is not an issue on a modern computer.

A final remark about memory consumption: the memory requirement of an explicit-state analysis is linear in the number of concurrently considered pipeline states. Therefore, one might expect the analysis to run out of memory in cases of state explosion. There are two reasons why this usually does not happen. The first reason is that the computation time is subject to the same linear growth and may become unacceptable before the analysis runs out of memory. The second reason, which is probably more important, is related to the way in which abstract execution traces are computed by explicit-state pipeline analyzers. They traverse the reachable state space of the pipeline model in depth-first order (see Figure 5.6) and thereby avoid handling large sets of pipeline states. While this approach limits the memory consumption to an acceptable level, it has no effect on the computation time which therefore remains the bottleneck in cases of state explosion.

We conclude this performance comparison by comparing the sizes of the BDDs for representing sets of states in order to illustrate the required computational effort. To

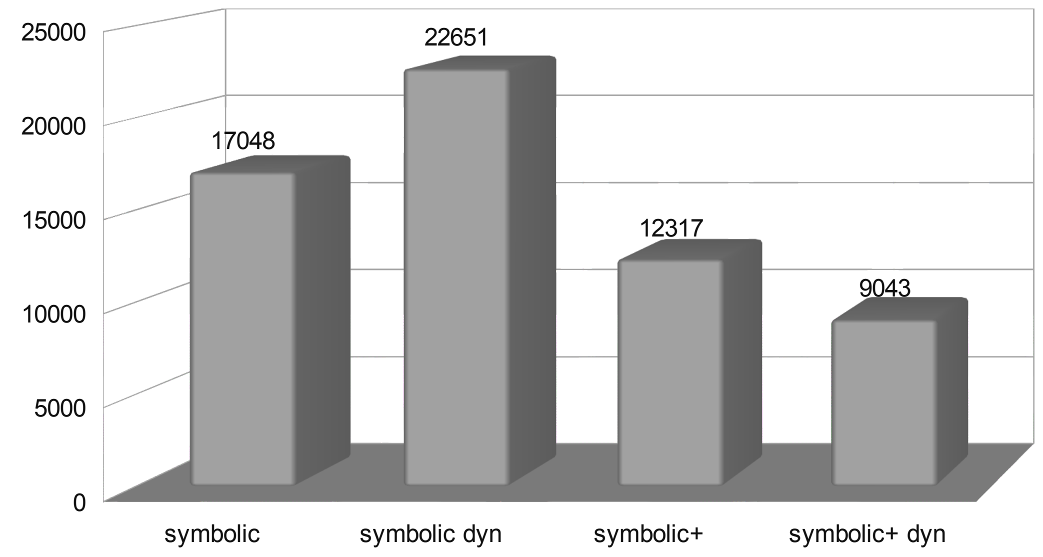


**Figure 7.11.:** Maximum BDD sizes for Dhrystone with 7 splits per fetch.

this end, Figure 7.10 shows the average number of BDD nodes for the Dhrystone benchmark in the configuration with 7 state splits per cycle. Again, the numbers underline the effectiveness of our optimizations. However, a comparison with the performance results of Table 7.2 also reveals that the cost for the symbolic translation of sets between basic blocks is significant. Otherwise, the symbolic+ implementation should have outperformed the plain symbolic implementation because of the smaller BDDs. Moreover, it is apparent that even a small reduction of BDD sizes can lead to a significant gain in performance.
The maximum BDD sizes are given in Figure 7.11. Most of the numbers meet our expectations but the result for the basic symbolic implementation with dynamic reordering is much higher than expected. This illustrates the heuristic nature of dynamic reordering which can explore permutations in the variable order that locally reduce the BDD size but may lead to a blowup at a later point of the analysis.

### 7.2.5. Typical Cache Access Patterns

The semi-symbolic domain for interfacing abstract caches (see Section 6.5) has not been implemented, yet. Anyway, this domain cannot be reliably evaluated with the deterministic TriCore 1 pipeline model of Section 7.2.3. In Section 7.2.4 we modified that model to lose precision on the latencies of memory accesses in order to force the coverage of a larger state space. This setup is not fit for evaluating the integration with the abstract cache analysis since it leads to a misrepresentation of the distribution of cache accesses. To obtain meaningful results we use the commercial, explicit-state pipeline model of the Motorola/Freescale PowerPC 755 [The04, Abs00] instead. This processor is equipped with separate data and instruction caches. However, the processor's bus unit – which is part of the abstract pipeline model – can only issue a single memory access (either an instruction fetch or a data memory access) per cycle. It can therefore be analyzed by the domain of Section 6.5 which requires a special treatment of pipeline state variables that encode memory accesses. Analyses with the pipeline model of the PowerPC 755 routinely reach numbers of states that are large enough to asses the efficiency of the semi-symbolic domain of Section 6.5.

### Evaluation of Avionics Tasks

We evaluate 6 tasks of a commercial, safety-critical real-time software of the avionics domain.[6] The tasks have been fully unrolled and annotated to avoid serious state explosion. The employed annotations specify ranges for register contents at selected program points to improve the precision of the value analysis and thereby reduce the reachable state space of the pipeline model. Note that full unrolling is not feasible for all software but required to obtain results with explicit-state implementations of very complex pipeline models. Otherwise, the analysis would not terminate in acceptable time because of state explosion. Even with unrolling and annotations the analysis often reaches up to several thousand pipeline states per cycle.
To evaluate the efficiency of the semi-symbolic domain we analyze the 6 tasks with an instrumented pipeline model of the PowerPC 755. The instrumentation code prints the following information for each access into a cached memory area:

1. Type of access, i.e., instruction or data.
2. Address and context of the currently analyzed basic block.
3. Cycle count since start of current basic block.
4. Accessed address or address range.

The raw instrumentation data is post-processed to obtain estimates of the expected number of partitions – the number of tuples of pipeline BDD and abstract cache state – and the expected sharing, i.e., the number of pipeline states that are encoded in the pipeline BDD of a single partition. To this end, we collect all accesses with equal access type, basic block address, analysis context, and cycle count. Symbolic pipeline analysis explores the model's state space cycle-wise in breadth-first order. Hence, all accesses in the same set are issued from pipeline states in the same exploration layer. We partition the sets depending on the accessed addresses to obtain the number of different memory accesses from the same layer. The results are listed in the following tables. For each task (numbered $t_1, \ldots, t_6$) the first row gives the results

[6]Closed source and confidential.

for instruction cache accesses, whereas the second row reports the same information for data cache accesses.

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ |
|---|---|---|---|---|---|---|
| I-Cache | 2.19 | 1.51 | 1.94 | 2.03 | 2.13 | 1.52 |
| D-Cache | 1.38 | 1.11 | 1.29 | 1.35 | 1.35 | 1.02 |

**Table 7.7.:** Average number of partitions per cycle.

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ |
|---|---|---|---|---|---|---|
| I-Cache | 17.24 | 35.35 | 25.49 | 25.61 | 35.63 | 19.96 |
| D-Cache | 10.82 | 28.07 | 20.76 | 25.94 | 25.61 | 7.87 |

**Table 7.8.:** Average number of pipeline states per partition.

The first table – Table 7.7 – shows the average number of partitions per cycle. It corresponds directly to the expected average number of partitions of an element of the semi-symbolic domain $\mathcal{D}^h$. The average number of partitions over all analyzed tasks is roughly 2. Table 7.8 shows the average sharing, i.e., the number of pipeline states that can be represented by the BDD of a single partition because the states access the same cached memory address. The average sharing over all 6 tasks is roughly 22. By multiplication with the number of partitions we obtain an estimate of the average number of pipeline states per cycle; it ranges between 14 and 70 states with an average of 44.

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ |
|---|---|---|---|---|---|---|
| I-Cache | 42 | 7 | 42 | 42 | 42 | 10 |
| D-Cache | 6 | 2 | 6 | 6 | 6 | 2 |

**Table 7.9.:** Maximum number of partitions per cycle.

More interesting than the average numbers are the *maximum* numbers of partitions and pipeline states per cycle. The maximum number of partitions per task is shown in Table 7.9. It bounds the number of cache states that are enumerated by the semi-symbolic domain $\mathcal{D}^h$. We see that the number of partitions stays fairly small. We need at most 42 partitions for disambiguating instruction cache accesses and 6 partitions for disambiguating data cache accesses. In contrast to the small number of partitions, the maximum number of pipeline states per cycle can be quite large as shown by the results in Table 7.10. More than 8000 pipeline states can be represented by the BDD of a single partition.

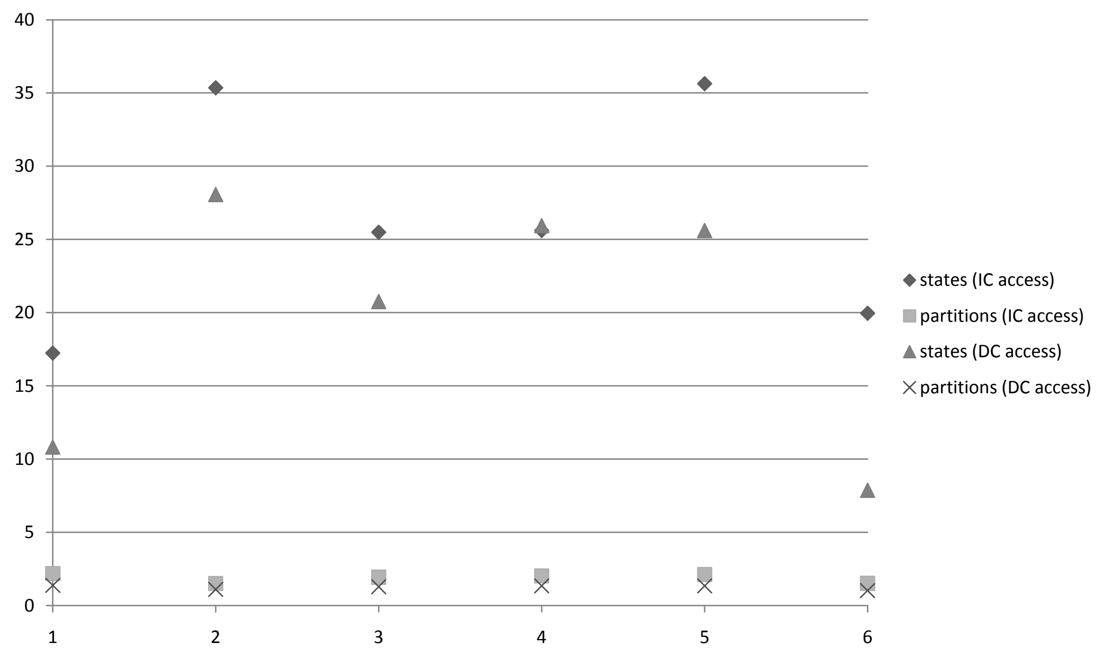


**Figure 7.12.:** Relation between the average numbers of pipeline states and partitions.

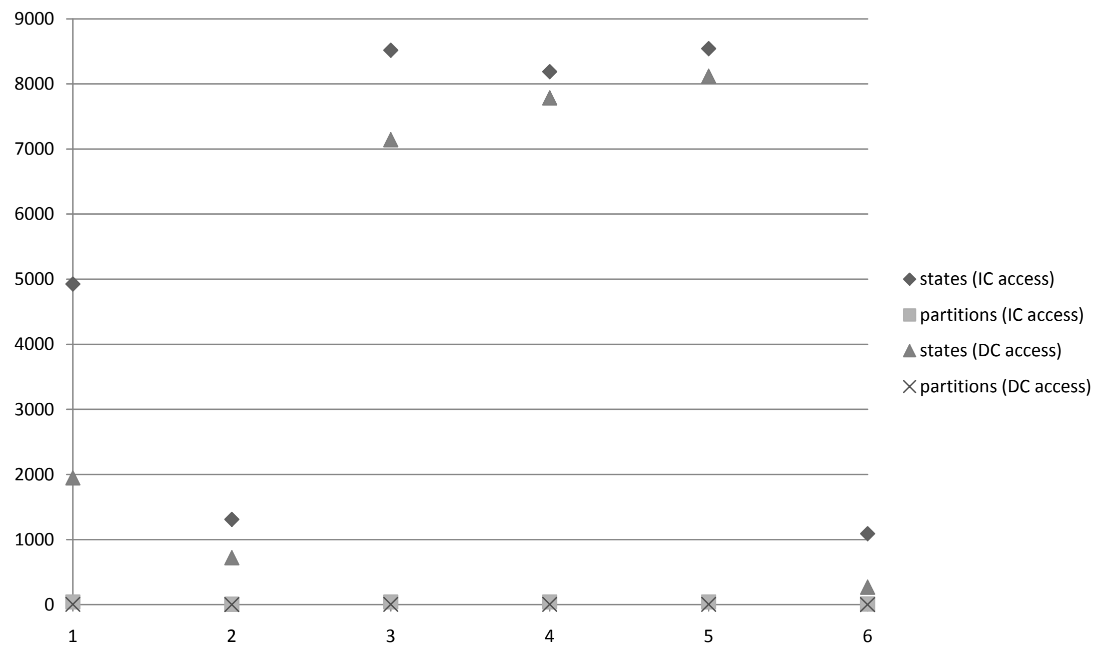


**Figure 7.13.:** Relation between the maximal numbers of pipeline states and partitions.

| | $t_1$ | $t_2$ | $t_3$ | $t_4$ | $t_5$ | $t_6$ |
|---|---|---|---|---|---|---|
| I-Cache | 4927 | 1311 | 8519 | 8190 | 8544 | 1091 |
| D-Cache | 1947 | 720 | 7140 | 7783 | 8115 | 268 |

**Table 7.10.:** Maximum number of pipeline states per partition.

### Interpretation of the Results

The semi-symbolic domain $\mathcal{D}^h$ of Section 6.5 operates on tuples of pipeline BDDs and abstract cache states. These tuples are called partitions. The domain is most efficient if the number of partitions is small. It may outperform a purely explicit-state approach if a large number of pipeline states can be encoded in the BDD of a single partition. The results of the experiments indicate that the number of partitions can indeed be expected to stay small. At the same time, the number of pipeline states in a single partitions can be quite large.

The relation between the average and maximum numbers of pipeline states and partitions is also displayed in Figure 7.12 and Figure 7.13, respectively. A large distance between two corresponding points (related either to instruction cache (IC) accesses or data cache (DC) accesses) indicates a favorable relation. Besides showing that this relation is indeed often favorable, the charts also evidence that the behavior for instruction cache and data cache accesses is usually quite similar. To conclude, one can say that the typical pipeline-cache relation ranges between 1 : 1 and 8544 : 1, with an average of 22 : 1. These numbers hold under the assumption that the analysis maintains maximum cache precision. The domain allows higher numbers of pipeline states per partition if caches are joined more aggressively. Larger numbers of pipeline states per partition can also be expected when the analysis encounters cases of imprecise information, e.g., about memory accesses.

## 7.3. Lessons Learned from Implementation

During the implementation of the prototype and by running the analyzer we learned that using symbolic methods for pipeline analysis not only improves the performance of the analysis in cases of state explosion. It also brings about further implications that are of practical relevance for developers of static WCET analyzers. These implications concern the required effort for specifying the non-determinism in abstract pipelines, the debugging of pipeline models, and the interfacing with an alternative path analysis which operates on so-called pipeline state graphs. This section presents our experience with these implications and gives an assessment of their impact on the application of the method in practice.

## 7.3.1. Dealing with the Non-Determinism

A significant advantage of using a symbolic image computation engine for the state traversal is the fact that it frees the designer of the abstract model from considerations about how to handle the non-determinism of the abstract pipeline. Explicit-state pipeline models are specified in terms of a non-deterministic transition function. The function is invoked for each state in the current set and each invocation produces one or several outgoing states. During the implementation of this function, the designer must decide where he wants to split the incoming state in cases of imprecise inputs, thereby producing several outgoing states. Since there may be several sources of non-determinism that have to be considered for a single transition, the state splitting becomes complex and error prone. In contrast, abstract pipeline models for the symbolic pipeline domain of Chapter 6 are specified as deterministic FSMs. The non-determinism of the corresponding abstract pipeline is expressed via additional inputs. In cases of imprecise information, all reachable successor states are considered automatically by the image computation in the algorithm of Figure 6.4.
We already pointed out in Section 6.6 that a transfer function for explicit-state pipeline analyses could in principle also be generated from a formal specification. While such work is already under way [SP10] the presented symbolic implementation still has the advantage of relying on established, mature tools and a huge body of research in the areas of hardware design and hardware verification. Our experience with the implementation of the TriCore 1 model (see Section 7.2.1) shows that these tools are easy to use and that the specification of a model is quite compact and readable (see Appendix A.1 for an example) which eases both design and maintenance.

## 7.3.2. Debugging and Pipeline State Graphs

Explicit-state pipeline analyses in the `aiT` tool provide a visualization that shows the evolution of the pipeline states in the analysis of a certain basic block in a certain context. A pair of predecessor and successor states in two subsequent sets is explicitly connected by an edge. This visualization makes it easy to understand the taken transitions and the edge information comes for free with explicit state traversal. In contrast, the symbolic pipeline domain of Chapter 6 does not enumerate the relations between two subsequent sets of pipeline states. The relations are given implicitly by the combined symbolic transition relation of the pipeline model and the program. Hence, producing a similar visualization is not possible, at least not in an efficient way. As a consequence, debugging pipeline models – in particular during the development of a pipeline model – can become harder compared to explicit-state pipeline domains.
We found that this issue is indeed relevant in practice. Certain bugs in the design of a pipeline model can lead to unexpected state explosion. Without a visualization of the predecessor-successor relationship, these bugs become very hard to fix because it is much more difficult to identify the original state that causes the unexpected state explosion.

The problem of debugging the state traversal is not only related to the missing visualization of explicit predecessor-successor edges. Debugging a symbolic pipeline analyzer is always difficult if a problem appears in a transition that involves a large number of states. We found that it is infeasible to compute an explicit visualization of symbolic pipeline state sets that are larger than about 1000 states. The equivalent BDD representation can be stored efficiently as a graph, but computing an acceptable graph layout is very expensive and the BDD is usually incomprehensible. However, the use of a formal specification of the model and the fully automatic state splitting (see Section 7.3.1) rather decreases the initial number of bugs in the model. Also, certain properties of the model specification could be checked automatically by a model checker (the VIS model checker, in our case) to further eradicate bugs from the model.
Finally, let us get back to the explicit successor-predecessor relation that is computed by explicit-state pipeline analyses. This information can also be used to generate so-called *pipeline state graphs*. Based on these graphs one can compute tighter WCET bounds by finding the worst-case path directly as a sequence of pipeline states [Ste10]. This method avoids the loss of precision that is caused by considering infeasible combinations of abstract basic block executions when computing the worst-case path only on the basic block graph. Of course, symbolic pipeline analyzers cannot efficiently generate pipeline state graphs because they do not explicitly enumerate the state transitions. However, this inability should not be overrated. Worst-case path analysis on pipeline state graphs works only on well-behaved programs. These programs are not the primary target for symbolic pipeline analysis because their WCET analysis usually does not suffer from state explosions. Finally, in the presence of serious state explosions explicit-state pipeline analyzers cannot generate the pipeline state graph either.

## 7.4. Summary

The symbolic algorithms for pipeline analysis that have been presented in Chapter 6 have been implemented in the `aiT` WCET tool. They are instantiated by a Verilog model of an abstract pipeline and a set of functions that describe how program information from the supergraph restricts the feasible transitions of the model. A first model has been implemented for the Infineon TriCore 1 processor core. Experiments with this model show that the implicit handling of sets of pipeline states alleviates the state explosion problem in WCET analysis. In cases of state explosion, the symbolic implementation easily outperforms the legacy explicit-state implementation. The experiments also show that the program decomposition optimization is indispensable for analyzing serious programs. The symbolic analysis is fully automatic. The problem of optimizing the variable order to reduce BDD sizes can be handled by existing dynamic variable ordering heuristics.
The integration with an abstract cache analysis has not been implemented. The performance of the semi-symbolic domain of Section 6.5 was assessed using an

instrumented explicit-state pipeline model of the PowerPC 755 instead. Data obtained from analyses with the instrumented model show that the semi-symbolic domain typically only enumerates very few cache states. At the same time, the number of pipeline states that can be handled implicitly in the same BDD can be quite large. It is likely that in cases of state explosion this domain will enable a similar performance improvement as the purely symbolic pipeline analysis without caches.

# CHAPTER 8

# Conclusion

Safe and precise WCET analysis must consider the variations in computation time that are caused by hardware features like pipelines and caches. Caches allow for abstract representations that over-approximate (potentially large) sets of concrete cache states and can be handled efficiently by static analyses. Similarly efficient abstract representations for pipelines are not known. Abstract pipeline semantics is commonly computed by large, non-deterministic FSMs and a substantial part of their state space must be covered by static WCET analysis. Because of the existence of timing anomalies and domino effects it is unsafe to simplify the state traversal by making local assumptions about the worst-case. Covering all reachable pipeline states often leads to the so-called state explosion problem in WCET analysis; the number of pipeline states that have to be considered for the same execution cycle grows too large. In such cases explicitly enumerating all states, which is required by state-of-the-art implementations, becomes infeasible due to memory and – more importantly – computation time limits.

Similar scalability problems are well-known from the area of hardware model checking. Symbolic methods based on BDDs have improved the applicability of model checking because they admit both an implicit encoding of the transition system and of analysis information, like sets of states in state traversal, and thereby avoid the explicit enumeration of these sets. This success story sparked our interest in using symbolic methods for WCET analysis. In the preceding chapters we described the integration of a symbolic state traversal engine into a static WCET analysis tool and detailed the arising problems and the solutions that we have found. This last chapter summarizes the results and contributions of our research and presents an outlook of industrial applications and directions for future research.

## 8.1. Results and Contribution

We present a novel symbolic approach for computing the abstract semantics of pipelines for WCET analysis. The approach admits an implicit representation of sets of pipeline states and avoids the – sometimes infeasible – enumeration of pipeline states. Thus, it can improve the performance of static WCET analysis in cases of state explosion without compromising the soundness of the method. The approach has been implemented in the `aiT` WCET tool using the pipeline model of the Infineon TriCore 1, a real-life processor that is commonly used for safety-critical automotive applications. The implementation has been evaluated on benchmarks and an engine control application from the automotive domain. The obtained results confirm that the use of symbolic methods for pipeline analysis can alleviate the state explosion problems in static WCET analysis. The improved performance can be explained by the following advantages:

1. Symbolic pipeline analysis computes traces of sets of pipeline states. Each set contains states that are reached in the same execution cycle relative to the beginning of a basic block. We observed that the states in such sets agree in most attributes and often differ only in few bits, e.g., in the possible values of a latency counter.[1] This observation agrees with the intuitive insight that pipeline states which execute the same instructions should not be too different. Hence, there is a large potential for sharing information that is invariant between the pipeline states. This potential is exploited in the BDD representation to avoid redundant computations in the state traversal.

2. During the fixed point iteration the analysis frequently computes the union of sets (at control flow joins) and checks the equivalence of sets (to determine whether a fixed point has been reached). Both operations have efficient BDD implementations which also exploit sharing and avoid redundant computations.

However, these advantages are of practical relevance only if the BDDs do not grow too large. Computations with large BDDs are quite expensive and this would render the analysis infeasible. Note that the BDD size constraint affects not only the representation of sets of pipeline states but also the implicit representation of the transition system of the pipeline model. Besides the already discussed potential for sharing, BDD sizes are further affected by

1. the size of the pipeline model in terms of the number of state variables, and

2. the ordering of the state variables.

Our results show that existing automatic variable reordering methods based on heuristics are sufficient for pipeline analysis. Since a fully automatic method is

[1]In some cases the variability between the pipeline states can be larger, e.g., for incoming states from different predecessor blocks. Nonetheless, the observation holds for the majority of pipeline states and execution cycles.

preferable in an industrial context we did not investigate whether anything can be gained from further manual optimization. Instead, we focused on the problem of model size which turns out to be very important for scaling, not only to realistic models, but also to industrial-level programs. To this end, we have presented optimizations that are specific to a certain pipeline model as well as several target independent optimizations. In particular the target independent optimizations – address compression and program decomposition – are crucial for scaling the analysis to large programs. Note that the program decomposition optimization, although target independent, is specific to the design of our static WCET analysis. It relies on the fact that the analysis may lose the correspondence between execution traces at basic block boundaries. This possibility of losing information to reduce the complexity of the analysis is a significant advantage of our method over the model-checking approach of Larsen et. al. [DOT$^+$10], which is the most closely related work.

Serious research on pipeline analysis cannot ignore the influence of caches. To obtain safe and precise WCET bounds for architectures with caches it is indispensable to model the interdependence of pipeline and cache states. Regarding each component in isolation would either be unsafe or lead to gross overestimations, depending on the assumptions about the behavior of the other component. Therefore, it is common practice to combine abstract pipeline and cache domains within a single abstract interpretation of the hardware [WEE$^+$08]. The exchange of information between both domains follows the general operation of the hardware; the pipeline domain sends memory access ranges to the cache domain which returns latency information (hit, miss, or both). The theory of abstract interpretation guarantees that the result of this combination – the reduced product of the domains – is sound [CC77].

Combining existing cache domains with our symbolic pipeline domain is difficult. A straightforward implementation following existing designs (e.g., as detailed by Thesing [The04]) would be inefficient because it would enumerate the – otherwise implicitly stored – pipeline states. In this work we have presented a more efficient solution which only enumerates the different memory ranges that can be accessed in a certain execution cycle. To this end, we constrain the ordering of state variables which encode cached memory accesses and exploit the ability of cache analysis to deal with ranges instead of precise addresses. Experimental data indicates that the number of different memory ranges that are accessed from the same execution cycle (relative to the beginning of a basic block) is usually small. However, our solution also allows to enumerate fewer but coarser ranges. Thus, it is possible to trade precision for efficiency, something that cannot be done with explicit-state pipeline analyses. Finally, note that the alternative of including caches directly into the symbolic representation would also be inefficient. Even abstract caches [Fer97] have much larger state representations than pipeline models.[2] The huge number of required state variables would lead to very large BDDs.

[2]An estimate for the required number of state bits for representing the state of a realistic cache is given in Section 6.5.1.

To conclude, let us summarize our major contributions:

1. We describe a *novel* and *efficient* integration of a symbolic state traversal engine into an abstract interpretation based static analysis framework. Model size is kept relatively small since arithmetic is handled by the value analysis. Further, no global cycle counter is required within the symbolic representation.

2. The integration comprises a *generic* solution for the interaction between a symbolic domain and other static analyses. The interaction is based on an explicit program representation as a control flow graph and supports advanced context-sensitive analysis. This enables interprocedural analysis and the precise analysis of loops. Furthermore, we also show how the symbolic pipeline domain can be combined with an abstract cache domain to arrive at an efficient, combined analysis of the hardware behavior.

3. We present a solution to scale the symbolic analysis to *realistic* program sizes. State traversal costs only depend on the size of the pipeline model, not on the program size or the number of analysis contexts.

4. We give experimental *evidence* showing that symbolic state traversal for pipeline analysis is feasible in practice. Since BDD representations are sensitive to the size of the employed models, it is crucial that we are able to keep the size of realistic pipeline models within acceptable limits. Further, our experimental data indicate that the built-in automatic variable reordering heuristics of a state-of-the-art model checker work well for our application.

## 8.2. Industrial Application and Future Research

The prototype implementation of Chapter 7 demonstrates that the symbolic approach for pipeline analysis is feasible in practice. However, the implementation is solely focused on improving the worst-case performance of static WCET analysis. That is, performance is only improved if state explosions occur. The improved scalability in terms of covered states is paid for by an increased sensitivity with respect to model size and an unsatisfactory average-case performance. This is due to the fact that computing a single cycle update with BDDs has much higher basic cost. This cost is amortized only if many similar states can be updated for the same cycle. While state explosion can render an explicit-state analysis infeasible, such events occur only in relatively few cycles. Paying a higher basic cost for all remaining cycles is impractical, if not to say annoying, in practice. For analyses in which no state explosions occur the analysis times can increase from a few seconds to several hours, compared to an explicit-state implementation. In this section we present several ideas for improvement. Solutions are certainly needed for applying the method in general purpose, industrial-strength WCET analyzers. Thus, the following ideas would also make interesting subjects for future research.

### 8.2.1. Hybrid Pipeline Analyzers

One way for improving average performance would be to rely on an explicit state representation for as long as the number of concurrently considered states ranges below a certain threshold. Such a hybrid analyzer requires that the transition relations for the explicit-state and the symbolic-state computations are equivalent. A manual transformation, e.g., as performed for the implementation of the prototype of Chapter 7, is not acceptable in an industrial context. The development of pipeline models would become more difficult and there is also a high probability of introducing inconsistencies between the two implementations. Hence, it would be necessary to generate both implementations from a common specification.
A hybrid implementation could perform explicit state traversal for as long as relatively few hardware states have to be considered. As soon as the analysis runs into a state explosion, the current set of states could be converted into a symbolic representation and symbolic state traversal could be used until the current set of states decreases below a certain threshold. Converting sets of states between compatible explicit- and symbolic-state analyses can be implemented in a straightforward manner. The main problem is efficiency: performance will only be acceptable if the conversion is performed on small sets. This means that simply translating to a symbolic representation *after* a state explosion has happened, cannot be recommended. Instead, the analysis must be able to go backwards to the last state *before* the state explosion took place. At this point, the costs of translating between the representations should be acceptable.
One last problem that requires consideration are the different state traversal strategies – depth-first versus breadth-first traversal. The different strategies of the two approaches prevent switching the representation at arbitrary execution cycles. The only safe points for switching are the entry points of basic blocks, right before the block-local state traversal starts. However, the performance degradation caused by this restriction might be insignificant in practice.

### 8.2.2. Alternative Symbolic Representations

BDDs are probably the best known and most widespread symbolic representation. However, many more symbolic representations exist, some of which are tailored to specific applications. Examples are Boolean function vectors [GB03], multi-valued decision diagrams (MDDs) [CDE01], or *zero-suppressed binary decision diagrams* (ZDDs) [Min93]. The latter are interesting in the context of symbolic pipeline analysis because ZDDs are more efficient for handling sparse sets[3] and the representation can also be used for symbolic state space exploration [YHTM96]. As we already pointed out in the previous section, state explosions in WCET analysis only occur at relatively few points during an analysis run. Hence, dealing more efficiently with sparse

[3]This is a consequence of the modification of the BDD first reduction rule [Mis01]. For BDDs a node is removed if both edges point to the same node (see the description of the elimination rule in Section 4.2.1). For ZDDs, a node is removed if its 1-edge points to the terminal node 0.

sets improves the average case performance of the symbolic computations. Dealing efficiently with sparse sets may even be a key to scaling the approach to more complex pipeline models. As the complexity of the pipeline model increases, so does its state space. The average set of hardware states that is considered by WCET analyses with this model increases, too. However, the *percentage* of considered pipeline states with respect to the overall number of states of the model usually *decreases* with growing model complexity. Hence, the constructed sets of pipeline states are more sparse for larger models and more efficiency might be gained from using a representation that is better suited for representing sparse sets.

### 8.2.3. Scaling to Larger Pipeline Models

The scalability problem of pipeline analysis grows with the size and complexity of the underlying pipeline model. The use of symbolic methods together with a set of suitable optimizations solves the problem for pipeline models of medium complexity. The Infineon TriCore 1 model that we considered for the prototype implementation in Chapter 7 is such a model. In fact, it is one of the smaller pipeline models where an explicit-state pipeline analysis can become infeasible in practice due to state explosions. Simpler pipelines, e.g., the ARM7 [ARM01] pipeline, are either free of timing anomalies[4] or the maximal number of states for each cycle is small enough to allow for an explicit enumeration. It is reasonable to assume that larger, more complex pipeline models are always prone to timing anomalies and probably also to domino effects (see Section 5.3.2). Consequently, it would be most interesting to scale the presented symbolic techniques to such large models. To this end, the following problems remain to be solved:

1. The representation of a pipeline model with deeper or more parallel pipelines needs more state variables for storing instruction references. The number of state variables is further increased if features like dynamic branch prediction or speculation have to be modeled.

2. The program decomposition optimization for efficiently handling large programs (see Section 6.4.4) may be not effective enough for larger pipeline models. The reason is that the size of the employed enumeration window is proportional to the size of the pipeline model. The more instructions fit into the window, the more state variables are needed for representing instruction references.

There are several ways for tackling these problems: new target-specific optimizations could be designed for implementing advanced features in very compact ways. The range of relevant program information for program decomposition could be further limited by arguing about architectural properties. Certain expensive features, e.g., branch target buffers, could be implemented in a more abstract manner. The resulting

[4]So far, the absence of timing anomalies could not be proven for any non-trivial pipeline model. However, in the case of the very simple ARM7 any delay always stalls the whole pipeline. Hence, one can immediately conclude that the local worst-case is identical to the global worst-case.

lack of information, e.g., about buffer contents, would force the analysis to explore more pipeline states in order to cover all possible behaviors. However, the number of pipeline states is not an issue in symbolic pipeline analysis. Performance depends only on the size of the involved BDDs and one major contributing factor is the number of state variables of the pipeline model.

## 8.3. Outlook

The presented work establishes fundamental techniques for integrating symbolic methods efficiently into a modular static WCET analysis. The use of symbolic methods solves an important scalability problem in WCET analysis but it also brings about new problems. Besides the already discussed questions of scaling to larger pipeline models and reducing the average costs of the computation, the significant effort for porting existing pipeline models also cannot be ignored.
However, the trend towards more complex soft- and hardware in embedded systems tends to exacerbate the inherent scalability problems in WCET analysis. At the same time, recent and upcoming industrial certification standards, e.g, ISO-26262 [ISO09] and DO-178C[5], recommend the use of static analysis techniques for verifying safety-critical properties, such as the adherence to execution time constraints. In the near future, unsound approaches may no longer be accepted for the most critical applications. Thus, we expect that the demand for sound but more efficient WCET tools increases. This demand is addressed by our work. At the time of writing, we are not aware of feasible and sound alternatives for analyzing complex software running on embedded processors with non-trivial pipelines.

[5]To be published.

# APPENDIX A

# Fetch unit model

The following code excerpts are taken from the implementation of the pipeline model of the Infineon TriCore 1 as described in Chapter 7. The excerpts show the model's fetch unit which manages the prefetch buffer and the dispatch of instructions into the two major pipelines. This part of the model makes a good example for illustrating the interplay of the transition relations of the abstract pipeline model and the program. The state of the fetch unit is updated according to the transition relation of the pipeline model as given by the Verilog code in Appendix A.1. When executing non-linear code sequences, i.e., when redirecting the fetch address to the target of a branch or call instruction, the next-state is further influenced by the program transition relation. The program transition relation also comes into play when the contents of the prefetch buffer are updated; it restricts the possible buffer contents to the types and sizes of the fetched instructions. The construction of the program transition relation is based on C++ code. Examples are given in Appendix A.2.
The Verilog code in Appendix A.1 shows the complete abstract fetch unit model which defines the operation of the fetch unit independent of any program. It is written in the Verilog subset that is accepted by the VIS system [BHSV+96]. The C++ code for constructing the relevant part of the program transition relation in Appendix A.2 shows only a part of the function `buildRelationsForFetchDataAndDecodeTargets` and several supporting functions. It is meant to be sufficient to illustrate the implementation principle. The relations are constructed from BDD primitives using a C++ wrapper type `Cf` (which stands for *characteristic function*) that we designed according to our needs. This data type features handy operators that correspond to unary and binary BDD operations, e.g., `!` for negation, `+` for disjunction, and `*` for conjunction. The implementation is put on top of the VIS API. The low-level BDD operations are carried out by the CUDD library [Som09].

## A.1. Unit Transition Rules

Here is the Verilog code that defines the transition system of the fetch unit. The code can be compiled into the BDD representation of the symbolic model transition relation by VIS. Hints for understanding the model are given on page 139.

Verilog Model, Page 1

```
module FetchUnit(clock,
                 target_ma,        // IN : branch/call/return target
                 target_ls,        // IN : jump/call/return target
                 state_ma,         // IN : ready for dispatch to INT pipeline
                 state_ls,         // IN : ready for dispatch to LS pipeline
                 MA_fetch_ack,     // OUT: INT pipeline dispatch done
                 LS_fetch_ack,     // OUT: LS pipeline dispatch done
                 dispatch_to_ma,   // OUT: INT pipeline, dispatched address
                 dispatch_to_ls,   // OUT: LS pipeline, dispatched address
                 address_to_fetch,// OUT: address to fetch from bus
                 trigger_fetch,    // OUT:
                 linecrossing,     // OUT: fetch crosses 32 byte line
                 bus_data_valid,   // IN : fetched data is valid
                 bus_stall         // IN : bus is busy
                 );

   // function for extracting pipe bit from fetch
   // data buffer depending on current buffer index
   function cur_pipe;
      input [0:2] idx;
      input [`FETCH_DATA_WIDTH] buffer;

      begin
         case (idx)
           0: cur_pipe = buffer[0];
           1: cur_pipe = buffer[2];
           2: cur_pipe = buffer[4];
           default: cur_pipe = buffer[6];
         endcase
      end
   endfunction

   // function for extracting width bit from fetch
   // data buffer depending on current buffer index
   function cur_width;
      input [0:2] idx;
      input [`FETCH_DATA_WIDTH] buffer;

      begin
         case (idx)
           0: cur_width = buffer[1];
           1: cur_width = buffer[3];
           2: cur_width = buffer[5];
           default: cur_width = buffer[7];
         endcase
      end
   endfunction

   // ---
   // in/out declarations
   // ---

   input clock;
   input [`INSTR_WIDTH] target_ma;
   input [`INSTR_WIDTH] target_ls;
```

Verilog Model, Page 2

```
    input state_ma; sig_stall_t wire state_ma;
    input state_ls; sig_stall_t wire state_ls;

    output MA_fetch_ack;
    reg MA_fetch_ack;
    initial MA_fetch_ack = 0;

    output LS_fetch_ack;
    reg LS_fetch_ack;
    initial LS_fetch_ack = 0;

    output dispatch_to_ma;
    reg [`INSTR_WIDTH] dispatch_to_ma;
    initial dispatch_to_ma = 0;

    output dispatch_to_ls;
    reg [`INSTR_WIDTH] dispatch_to_ls;
    initial dispatch_to_ls = 0;

    output address_to_fetch;
    reg [`INSTR_WIDTH] address_to_fetch;
    initial address_to_fetch = 0;

    output trigger_fetch;
    reg trigger_fetch;
    initial trigger_fetch = 0;

    wire linecrossing_input;
    get_linecrossing(linecrossing_input);
    output linecrossing;
    reg linecrossing;
    initial linecrossing = 0;

    input bus_data_valid;
    input bus_stall;

    // ---
    // internal declarations
    // ---

    reg [`INSTR_WIDTH] buffer_addr;
    initial buffer_addr = 0;

    reg [0:2] buffer_idx;
    initial buffer_idx = 4;

    reg [`FETCH_DATA_WIDTH] buffer_data;
    initial buffer_data = 0;

    wire [`INSTR_WIDTH] input_addr;
    get_addr(input_addr);

    wire [`FETCH_DATA_WIDTH] fetch_input_data;
    get_data(fetch_input_data);

    reg [`FETCH_DATA_WIDTH]  data;
    initial data = 0;

    reg start;
    initial start = 1;
```

Verilog Model, Page 3

```
    // ---
    // transitions
    // ---

    always begin if (! clock) begin
       if (trigger_fetch && (0 != address_to_fetch)) begin
          data = fetch_input_data;
          linecrossing = linecrossing_input;
       end

       else begin
          linecrossing = 0;
       end
    end end

    always begin if (clock) begin

       trigger_fetch = 0;

       if (start) begin // issue first fetch
          address_to_fetch = input_addr;
          trigger_fetch = 1;
          MA_fetch_ack = 0;
          LS_fetch_ack = 0;
          start = 0;
       end

       if (bus_data_valid) begin // merge and issue next fetch

          if (0 == buffer_idx) begin // nothing dispatched
             if (! bus_stall) begin
                // discard fetched data and issue the fetch again
                // to prevent the pipeline from stalling
                address_to_fetch = buffer_addr + 4;
                trigger_fetch = 1;
             end
          end

          else if (1 == buffer_idx) begin  // dispatched one halfword
             buffer_addr = buffer_addr + 1;
             buffer_idx = 0;
             buffer_data[0] = buffer_data[2];
             buffer_data[1] = buffer_data[3];
             buffer_data[2] = buffer_data[4];
             buffer_data[3] = buffer_data[5];
             buffer_data[4] = buffer_data[6];
             buffer_data[5] = buffer_data[7];
             buffer_data[6] = data[0];
             buffer_data[7] = data[1];

             if (! bus_stall) begin
                address_to_fetch = address_to_fetch + 1;
                trigger_fetch = 1;
             end
          end

          else if (2 == buffer_idx) begin // dispatched two halfwords
             buffer_addr = buffer_addr + 2;
             buffer_idx = 0;
             buffer_data[0] = buffer_data[4];
             buffer_data[1] = buffer_data[5];
             buffer_data[2] = buffer_data[6];
             buffer_data[3] = buffer_data[7];
```

Verilog Model, Page 4

```
            buffer_data[4] = data[0];
            buffer_data[5] = data[1];
            buffer_data[6] = data[2];
            buffer_data[7] = data[3];

            if (! bus_stall) begin
               address_to_fetch = address_to_fetch + 2;
               trigger_fetch = 1;
            end
         end

         else if (3 == buffer_idx) begin // dispatched three halfwords
            buffer_addr = buffer_addr + 3;
            buffer_idx = 0;
            buffer_data[0] = buffer_data[6];
            buffer_data[1] = buffer_data[7];
            buffer_data[2] = data[0];
            buffer_data[3] = data[1];
            buffer_data[4] = data[2];
            buffer_data[5] = data[3];
            buffer_data[6] = data[4];
            buffer_data[7] = data[5];

            if (! bus_stall) begin
               address_to_fetch = address_to_fetch + 3;
               trigger_fetch = 1;
            end
         end

         else begin // dispatched all four halfwords
            buffer_addr = address_to_fetch;
            buffer_data = data;
            buffer_idx = 0;

            if (! bus_stall) begin
               address_to_fetch = address_to_fetch + 4;
               trigger_fetch = 1;
            end
         end
      end // if (bus_data_valid)

      // clear old INT pipeline dispatch
      if (READY == state_ma)
         dispatch_to_ma = 0;

      // clear old LS pipeline dispatch
      if (READY == state_ls)
         dispatch_to_ls = 0;

      if ((READY == state_ma && READY == state_ls)) begin // dispatch

         if (READY == state_ma && (0 == cur_pipe(buffer_idx, buffer_data)) &&
              ((buffer_idx < 3) || ((buffer_idx == 3) &&
              (0 == cur_width(buffer_idx, buffer_data))))) begin

            case (buffer_idx)
              0: dispatch_to_ma = buffer_addr;
              1: dispatch_to_ma = buffer_addr + 1;
              2: dispatch_to_ma = buffer_addr + 2;
              default: dispatch_to_ma = buffer_addr + 3;
            endcase

            MA_fetch_ack = 1;
```

Verilog Model, Page 5

```
             if (cur_width(buffer_idx, buffer_data))
               buffer_idx = buffer_idx + 2;
             else
               buffer_idx = buffer_idx + 1;
          end

          else begin
             MA_fetch_ack = 0;
          end

          if (READY == state_ls && cur_pipe(buffer_idx, buffer_data) &&
              ((buffer_idx < 3) || ((buffer_idx == 3) &&
              (0 == cur_width(buffer_idx, buffer_data))))) begin

             case (buffer_idx)
               0: dispatch_to_ls = buffer_addr;
               1: dispatch_to_ls = buffer_addr + 1;
               2: dispatch_to_ls = buffer_addr + 2;
               default: dispatch_to_ls = buffer_addr + 3;
             endcase

             LS_fetch_ack = 1;

             if (cur_width(buffer_idx, buffer_data))
               buffer_idx = buffer_idx + 2;
             else
                buffer_idx = buffer_idx + 1;
          end

          else begin
             LS_fetch_ack = 0;
          end
       end // if ((READY == state_ma && READY == state_ls))

       if (target_ma) begin // handle redirect request from INT pipeline
          address_to_fetch = target_ma;
          trigger_fetch = 1;
          MA_fetch_ack = 0;
          LS_fetch_ack = 0;
          buffer_idx = 4;
          dispatch_to_ma = 0;
          dispatch_to_ls = 0;
       end

       else if (target_ls) begin // handle redirect request from LS pipeline
          address_to_fetch = target_ls;
          trigger_fetch = 1;
          MA_fetch_ack = 0;
          LS_fetch_ack = 0;
          buffer_idx = 4;
          dispatch_to_ma = 0;
          dispatch_to_ls = 0;
       end
    end end

 endmodule // FetchUnit
```

## Hints for Understanding the Model

The `ls_` and `ma_` signals of the fetch unit are connected to the load/store and multiply/accumulate (or integer) pipelines. The pipelines themselves are implemented as hierarchical Verilog modules containing further internal modules (decode, execute, and writeback stages). The helper functions `cur_pipe` and `cur_width` in line 19 and line 35 are used to extract information from our compact representation of the prefetch buffer (see Section 6.4.3).
The transition rules are split into transitions at the falling and rising edges of the clock signal, lines 123 to 132 and 134 to 298, respectively. At the falling edge of the signal, the fetch unit takes incoming data and integrates it into its prefetch buffer. However, most work is done at the rising edge of the clock signal where the fetch unit propagates the contents of the prefetch buffer into the pipelines depending on instruction type information. To this end, it checks how much data is still in the buffer and whether one or both of the pipelines are ready (using the signals `state_ma` and `state_ls`).

## A.2. Program Interface

The following C++ code illustrates the construction of the program transition relation from information that is extracted from the supergraph.

Program Relations, Page 1

```
typedef enum {
    MA = 0, // integer pipeline
    LS      // load/store pipeline
} pipe_t;

/* 2bits for encoding instruction data.
   [0] Pipeline: either MA (0) or LS (1)
   [1] Width   : whether instruction is 2 bytes (0) or 4 bytes (1) */
typedef enum {
    data_pipe_shift = 0,
    data_width_shift
} data_shift_t;

/* Encoding methods for various variables in the Tricore model. */
static Cf NEGEDGE() { return Cf("clock", 0, current); }
static Cf POSEDGE() { return Cf("clock", 1, current); }

/* FETCH unit related variables */
static Cf GLOBAL_trigger_fetch(int t, state_t s) {
    return Cf("GLOBAL_trigger_fetch", t, s); }
static Cf GLOBAL_address_to_fetch(CrlUnsigned a, state_t s) {
    return Cf(address(var_GLOBAL_address_to_fetch), a, s); }
static Cf GLOBAL_linecrossing(int lc, state_t s) {
    return Cf("GLOBAL_linecrossing", lc, s); }
static Cf FETCH_data(int dat, state_t s) {
    return Cf(data("FETCH.data"), dat, s); }

/* Encodes instruction data bits in 'd' and returns new 'd'. */
static int fetchDataHelper(int d, int idx, int width, pipe_t p) {
    return (d | ((p != MA) << (data_pipe_shift + 2 * idx)) |
            ((4 == width) << (data_width_shift + 2 * idx)));
}

/* Returns pipeline to which the single operation in stmt is dispatched. */
static pipe_t pipe(KFG_STATEMENT stmt)
{
    CrlOperation* op = stmt->single_operation();
    CrlSymbol genname = op->find_symbol_sym(symbol::genname());

    /* abs absb absdif absdifb absdifh absdifs absdifsh absh abss abssh */
    if (! strncmp(genname, "abs", 3))
        return LS;
    /* adda addiha */
    else if (sym_adda() == genname || sym_addiha() == genname)
        return LS;
    /* add addb addc addf addh addi addih adds addsca addscat addsh addshu addsu addx */
    else if (! strncmp(genname, "add", 3))
        return MA;
    /* and andandnt andandt and_cond andn andnort andnt andort andt */
    else if (! strncmp(genname, "and", 3))
        return MA;
    ...
}
```

Program Relations, Page 2

```

/* Build relation for retrieving data in fetch buffer and control-flow targets
   in decode units. */
void buildRelationsForFetchDataAndDecodeTargets(KFG_NODE node, KFG_POSITION pos)
{
    Cf cur, nxt;
    std::vector<StmtPosPair> collection = getCollection(node, pos);

    for (std::vector<StmtPosPair>::const_iterator first = collection.begin();
         first != collection.end(); ++first) {

        int d = 0; // encode 12 bits (width and pipe) for at most 4 instructions
        for (std::vector<StmtPosPair>::const_iterator follow = first;
             (((*follow).address() - (*first).address()) < 8);
             ++follow) {
            d = fetchDataHelper(d, ((*follow).address() - (*first).address()) / 2,
                                (*follow).width(), pipe((*follow).statement()));
        }

        // determine whether fetch crosses a 32 byte line
        int lc = ((*first).address() % 32) + 8 > 32 ? 1 : 0;

        int inum = map2inum(node, pos, first->address(), first->position());

        // encode relation for fetch data
        cur = GLOBAL_trigger_fetch(1, current)
            * GLOBAL_address_to_fetch(inum, current)
            * NEGEDGE();
        nxt = FETCH_data(d, next)
            * GLOBAL_linecrossing(lc, next);
        insertRelation(node, pos, (cur * nxt) + (! cur));
        ...
}
```

# Bibliography

[Abs00] AbsInt. http://www.absint.com/aiT/, 2000.

[AG08] Infineon Technologies AG. *TriCore 1, 32-bit Unified Processor Core, Core Architecture Manual V1.3.8*. Munich, Germany, January 2008.

[ALE02] Todd M. Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, 2002.

[Alt01] Altran. http://www.flexray.com, 2001.

[AMWH94] Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding worst-case instruction cache performance. In *Proceedings of the 15th IEEE Real-Time Systems Symposium, RTSS '94*, pages 172–181, 1994.

[ARM01] ARM. *ARM7 Technical Reference Manual*, April 2001.

[BBB03] Alan Burns, Guillem Bernat, and Ian Broster. A probabilistic framework for schedulability analysis. In Rajeev Alur and Insup Lee, editors, *EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2003.

[BBK$^+$06] Adam Betts, Guillem Bernat, Raimund Kirner, Peter Puschner, and Ingomar Wenzel. WCET Coverage for Pipelines. Technical report, University of York, 2006.

[BBN05] Guillem Bernat, Alan Burns, and Martin Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Computing*, 1(2):179–194, 2005.

[BCC$^+$03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.

[BCH$^+$91] Robert K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R.P. Kurshan, S. Malik, Alberto L. Sangiovanni-Vincentelli, E.M. Sentovich, T. Shiple, K.J. Singh, and H.Y. Wang. BLIF-MV: An interchange format for design verification and synthesis. Technical Report UCB/ERL M91/97, EECS Department, University of California, Berkeley, 1991.

[BCL$^+$94] Jerry R. Burch, Edmund M. Clarke, David E. Long, Kenneth L. McMillan, and David L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13:401–424, 1994.

[BCM$^+$90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: $10^{20}$ states and beyond. IEEE Comp. Soc. Press, 1990.

[BCP02] Guillem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 279–288, 2002.

[BCRS10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive WCET analysis. In *Proceedings of the 8th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'10, pages 35–46, Berlin, Heidelberg, 2010. Springer-Verlag.

[Ber02] Sergey Berezin. *Model Checking and Theorem Proving: a Unified Framework*. PhD thesis, Carnegie Mellon University, 2002.

[Ber06] Christoph Berg. PLRU cache domino effects. In Frank Mueller, editor, *6th International Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer (STTT)*, 9(5-6):505–525, 2007.

[BHSV$^+$96] Robert K. Brayton, Gary D. Hachtel, Alberto L. Sangiovanni-Vincentelli, Fabio Somenzi, Adnan Aziz, Szu-Tsung Cheng, Stephen A. Edwards, Sunil P. Khatri, Yuji Kukimoto, Abelardo Pardo, Shaz Qadeer, Rajeev K. Ranjan, Shaker Sarwary, Thomas R. Shiple, Gitanjali Swamy, and Tiziano

Villa. VIS: A System for Verification and Synthesis. In *Proceedings of the 8th International Conference on Computer Aided Verification, CAV '96*, pages 428–432, 1996.

[BLL$^+$96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal – a tool suite for automatic verification of real-time systems. In *Proceedings of the DIMACS/SYCON workshop on Hybrid systems III : verification and control*, pages 232–243, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[BLQ$^+$03] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie J. Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI '03*, pages 103–114, 2003.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison Wesley, 1990.

[CBM90] Olivier Coudert, Christian Berthet, and Jean Madre. Verification of synchronous sequential machines based on symbolic execution. In Joseph Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer Berlin / Heidelberg, 1990.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Los Angeles, California, 1977.

[CDE01] Marsha Chechik, Benet Devereux, and Steve Easterbrook. Implementing a multi-valued symbolic model checker. In *Proceedings of TACAS'01*, pages 404–419. Springer, 2001.

[CFG$^+$10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza Burguière, Jan Reineke, Benoît Triquet, Simon Wegener, and Reinhard Wilhelm. Predictability Considerations in the Design of Multi-Core Embedded Systems. *Ingenieurs de l'Automobile*, 807:26–42, 2010.

[CGH$^+$93] Edmund M. Clarke, Orna Grumberg, Hiromi Hiraishi, Somesh Jha, David E. Long, Kenneth L. McMillan, and Linda A. Ness. Verification of the futurebus+ cache coherence protocol. In *Proceedings of the 11th IFIP WG10.2 International Conference sponsored by IFIP WG10.2 and in* *cooperation with IEEE COMPSOC on Computer Hardware Description Languages and their Applications*, CHDL '93, pages 15–30, Amsterdam, The Netherlands, 1993. North-Holland Publishing Co.

[Che94] Szu-Tsung Cheng. Compiling Verilog into Automata. Technical report, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, 1994.

[Cor91] Intel Corporation. *i960 KA/KB Microprocessor Programmers Reference Manual*, 1991.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes.* PhD thesis, Université scientifique et medicale de Grenoble, 1978.

[CP00] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

[CP01] Antoine Colin and Isabelle Puaut. A modular & retargetable framework for tree-based WCET analysis. In *13th Euromicro Conference on Real-Time Systems, ECRTS '01*, pages 37–44. IEEE Computer Society, 2001.

[CRSS94] David Cyrluk, S. Rajan, Natarajan Shankar, and Mandayam K. Srivas. Effective theorem proving for hardware verification. In *Proceedings of the Second International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience, TPCD '94*, pages 203–222, London, UK, 1994. Springer-Verlag.

[DOT$^+$10] Andreas Engelbredt Dalsgaard, Mads Christian Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. METAMOC: Modular Execution Time Analysis using Model Checking. In *10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2010.

[EDN88] DSP Performance Benchmarks. In *EDN - Electronic Design, Strategy, News*, September 1988.

[Eng02] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis.* PhD thesis, Uppsala University, 2002.

[EPB$^+$06] Jochen Eisinger, Ilia Polian, Bernd Becker, Alexander Metzner, Stephan Thesing, and Reinhard Wilhelm. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In Matteo Sonza Reorda, Ondrej Novák, Bernd Straube, Hanna Kubátová, Zdenek Kotásek, Pavel Kubalík, Raimund Ubar, and Jiri Bucek, editors, *9th IEEE Workshop on Design & Diagnostics of Electronic Circuits & Systems (DDECS 2006)*, pages 15–20. IEEE Computer Society Press, 2006.

[Fal09] Heiko Falk. WCET-aware Register Allocation based on Graph Coloring. In *The 46th Design Automation Conference (DAC)*, pages 726–731, San Francisco / USA, July 2009.

[Fer97] Christian Ferdinand. *Cache Behavior Prediction for Real-Time Systems*. PhD thesis, Saarland University, 1997.

[FHL$^+$01] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and Precise WCET Determination for a Real-Life Processor. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

[FL10] Heiko Falk and Paul Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, pages 1–50, 2010.

[FMK91] Masahiro Fujita, Yusuke Matsunaga, and Taeko Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *EURO-DAC '91: Proceedings of the conference on European design automation*, pages 50–54, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[Fre01] Freescale. *MPC750 RISC Microprocessor Family User's Manual*, December 2001.

[GB03] Amit Goel and Randal E. Bryant. Set manipulation with boolean functional vectors for symbolic reachability analysis. In *Proceedings of Design Automation and Test in Europe, DATE '03*, pages 10816–10821, 2003.

[HAM$^+$99] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.

[HBL$^+$95] Yerang Hur, Young Hyun Bae, Sung-Soo Lim, Sung-Kwan Kim, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, and Chong-Sang Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3010 Case Study. In *IEEE Real-Time Systems Symposium*, pages 308–321, 1995.

[HHJ$^+$05] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis – the SymTA/S approach. In *IEEE Proceedings on Computers and Digital Techniques*, volume 152(2), pages 148–166, March 2005.

[HLS00] Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Worst-Case Execution Time Analysis for Digital Signal Processors. In *European Signal Processing Conference (EUSIPCO)*, 2000.

[HLTW03] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. In *Proceedings of the IEEE*, volume 91, pages 1038–1054, 2003.

[HWH95] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[ISO04] International Organisation for Standardization ISO. ISO 11898-4, Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication, 2004.

[ISO09] ISO 26262-WD. Road vehicles – Functional safety, 2009.

[ISY91] Nagisa Ishiura, Hiroshi Sawada, and Shuzo Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the IEEE International Conference on Computer Aided Design, ICCAD'91*, pages 472–475, 1991.

[JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture Verification by Compositional Model Checking. In *Proceedings of the 13th International Conference on Computer Aided Verification, CAV '01*, pages 396–410, 2001.

[KLFP02] Raimund Kirner, Roland Lang, Gerald Freiberger, and Peter P. Puschner. Fully automatic worst-case execution time analysis for Matlab/Simulink models. In *14th Euromicro Conference on Real-Time Systems, ECRTS '02*, pages 31–40. IEEE Computer Society, 2002.

[KP03] Raimund Kirner and Peter P. Puschner. Transformation of meta-information by abstract co-interpretation. In Andreas Krall, editor, *SCOPES*, volume 2826 of *Lecture Notes in Computer Science*, pages 298–312. Springer, 2003.

[LBJ$^+$94] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, and Chong-Sang Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. In *Proceedings of the 15th IEEE Real-Time Systems Symposium, RTSS '94*, pages 142–151, 1994.

[Liu00] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[LL73] C.L. Liu and Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the Association of Computing Machinery*, 20:64–61, 1973.

[LM09] Paul Lokuciejewski and Peter Marwedel. Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In *The 21st Euromicro Conference on Real-Time Systems, ECRTS '09*, pages 35–44, Dublin / Ireland, July 2009.

[LMR05] Xianfeng Li, Tulika Mitra, and Abhik Roychoudhury. Modeling control speculation for timing analysis. *Real-Time Systems*, 29(1):27–58, 2005.

[LMW96] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *IEEE Real-Time Systems Symposium*, pages 254–263. IEEE Computer Society, 1996.

[LNYY10] Mingsong Lv, Guan Nan, Wang Yi, and Ge Yu. Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software. In *Proceedings of the 31st IEEE Real-Time Systems Symposium*, 2010.

[LPY97] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1-2:134–152, Oct 1997.

[LRM04] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for software timing analysis. In *Proceedings of the 24th IEEE International Real-Time Systems Symposium, RTSS '04*, pages 92–103. IEEE Computer Society, 2004.

[LS98] Thomas Lundqvist and Per Stenström. Integrating path and timing analysis using instruction-level simulation techniques. In Frank Mueller and Azer Bestavros, editors, *LCTES '98*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1998.

[LS99a] Thomas Lundquist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, 1999.

[LS99b] Thomas Lundqvist and Per Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *Real-Time Systems*, 17(2-3):183–207, 1999.

[LS03] George Logothetis and Klaus Schneider. Exact high level WCET analysis of synchronous programs by symbolic state space exploration. In *Proceedings of Design Automation and Test in Europe, DATE '03*, pages 10196–10203, 2003.

[LSM03] G. Logothetis, K. Schneider, and C. Metzler. Exact low-level runtime analysis of synchronous programs for formal verification of real-time systems. In *Forum on Design Languages (FDL)*, Frankfurt, Germany, 2003. Kluwer.

[Mar98] Florian Martin. PAG – An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):46–67, 1998.

[Mar99] Florian Martin. *Generating Program Analyzers*. PhD thesis, Universität des Saarlandes, 1999.

[MAWF98] Florian Martin, Martin Alt, Reinhard Wilhelm, and Christian Ferdinand. Analysis of Loops. In Kai Koskimies, editor, *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 80–94, Berlin, 1998. Springer.

[McM92] Kenneth L. McMillan. *Symbolic Model Checking*. PhD thesis, Carnegie Mellon University, 1992.

[Met04] Alexander Metzner. Why model checking can improve WCET analysis. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV '04*, pages 334–347, 2004.

[Min93] Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proceedings of the 30th international Design Automation Conference*, DAC '93, pages 272–277, New York, NY, USA, 1993. ACM.

[Min04] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004.

[Mis01] Alan Mishchenko. An Introduction to Zero-Suppressed Binary Decision Diagrams. In *Proceedings of the 12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning*, 2001.

[Mot97] Motorola. *MPC750 RISC Processor User's Manual*, 1997.

[MT98] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1998.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[O'K00] Hugh O'Keeffe. IEEE-ISTO 5001-1999, The Nexus 5001 Forum Standard – providing the Gateway to the Embedded Systems of the Future. In *Proceedings of the Embedded Intelligence conference*, 2000.

[PS97] Peter P. Puschner and Anton V. Schedl. Computing maximum task execution times – a graph-based approach. *Real-Time Systems*, 13(1):67–91, 1997.

[RAB$^+$95] R. Ranjan, A. Aziz, R. Brayton, B. Plessier, and C. Pixley. Efficient BDD Algorithms for FSM Synthesis and Verification, 1995.

[Rad] Radio Technical Commission for Aeronautics. RTCA DO-178B. Software Considerations in Airborne Systems and Equipment Certification.

[Rei08] Jan Reineke. *Caches in WCET Analysis.* PhD thesis, Saarland University, 2008.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the 22nd ACM Symposium on Principles of Programming Languages*, pages 49–61. ACM Press, 1995.

[RLM02] Abhik Roychoudhury, Xianfeng Li, and Tulika Mitra. Timing analysis of embedded software for speculative processors. In *Proceedings of the 15th International Symposium on System Synthesis, ISSS '02*, pages 126–131. IEEE Computer Society, Oct 2002.

[RLP$^+$94] Byung-Do Rhee, Sung-Soo Lim, Chang Yun Park, Sang Lyul Min, Heonshik Shin, and Chong Sang Kim. Issues of advanced architectural features in the design of a timing tool. In *Proceedings of the 11th Workshop on Real-Time Operating Systems and Software*, pages 59–62, 1994.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47. IEEE Computer Society Press, 1993.

[Sch03] Jörn Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

[Sha89] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. In *IEEE Transactions on Software Engineering*, volume 15(7), pages 875–889, July 1989.

[SLH$^+$05] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Victor Jégu, Guillaume Borios, and Reinhold Heckmann. Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 21–24, 2005.

[SM10] Stefan Stattelmann and Florian Martin. On the Use of Context Information for Precise Measurement-Based Execution Time Estimation. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 64–76, 2010.

[Som09] Fabio Somenzi. *CUDD: CU Decision Diagram Package Release 2.4.2*, 2009.

[SP10] Marc Schlickling and Markus Pister. Semi-automatic derivation of timing models for WCET analysis. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*, pages 67–76. ACM, April 2010.

[Sta09] Stefan Stattelmann. Precise Measurement-Based Worst-Case Execution Time Estimation. Master's thesis, Saarland University, September 2009.

[Ste10] Ingmar Stein. *ILP-based Path Analysis on Abstract Pipeline State Graphs*. PhD thesis, Saarland University, 2010.

[Tar55] A. Tarski. A lattice theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

[The02] Henrik Theiling. ILP-based Interprocedural Path Analysis. In *Proceedings of the Workshop on Embedded Software*, Grenoble, France, 2002.

[The03] Henrik Theiling. *Control Flow Graphs for Real-Time System Analysis*. PhD thesis, Saarland University, 2003.

[The04] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[THY93] Seiichiro Tani, Kiyoharu Hamaguchi, and Shuzo Yajima. The complexity of the optimal variable ordering problems of shared binary decision diagrams. In Kam-Wing Ng, Prabhakar Raghavan, N. V. Balasubramanian, and Francis Y. L. Chin, editors, *ISAAC*, volume 762 of *Lecture Notes in Computer Science*, pages 389–398. Springer, 1993.

[Tid00] Tidorum. http://www.bound-t.com/, 2000.

[TM96] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[TSH$^+$03] Stephan Thesing, Jean Souyris, Reinhold Heckmann, Famantanantsoa Randimbivololona, Marc Langenbach, Reinhard Wilhelm, and Christian Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 625–632. IEEE Computer Society, 2003.

[WC10] Stephan Wilhelm and Christoph Cullmann. Integrating Abstract Caches with Symbolic Pipeline Analysis. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 36–43, 2010.

[WEE$^{+}$08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.

[Wei88] Reinhold Weicker. Dhrystone benchmark: rationale for version 2 and measurement rules. *SIGPLAN Notices*, 23(8):49–62, 1988.

[Wil04] Reinhard Wilhelm. Why AI + ILP is good for WCET, but MC is not, nor ILP alone. In *Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

[Wil05] Stephan Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *Proceedings of the 5th Intl. Workshop on Worst-Case Execution Time Analysis*, 2005.

[WKRP08] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter P. Puschner. Measurement-based timing analysis. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA '08*, volume 17 of *Communications in Computer and Information Science*, pages 430–444. Springer, 2008.

[WL04] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In William Pugh and Craig Chambers, editors, *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation, PLDI '04*, pages 131–144. ACM, 2004.

[WW07] Stephan Wilhelm and Björn Wachter. Towards symbolic state traversal for efficient WCET analysis of abstract pipeline and cache models. In *Proceedings of the 7th International Workshop on Worst-Case Execution Time (WCET) Analysis*, July 2007.

[WW09] Stephan Wilhelm and Björn Wachter. Symbolic state traversal for WCET analysis. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, pages 137–146, October 2009.

[YHTM96] Tomohiro Yoneda, Hideyuki Hatori, Atsushi Takahara, and Shin-ichi Minato. BDDs vs. Zero-Suppressed BDDs: for CTL symbolic model checking of Petri nets. In Mandayam Srivas and Albert Camilleri, editors, *Formal Methods in Computer-Aided Design*, volume 1166 of *Lecture Notes* *in Computer Science*, pages 435–449. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0031826.

[Zar01] Sorin Zarnescu. *TriCore Pipeline Behaviour & Instruction Execution Timing*. Infineon Technologies, 2001.

# Index