Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models

Dissertation

zur Erlangung des Grades eines Doktors der Ingenieurwissenschaften (Dr.-Ing.) der Naturwissenschaftlich-Technischen Fakultäten der Universität des Saarlandes

von
Diplom-Informatiker

Stephan Thesing

Saarbrücken
Juli 2004

Tag des Kolloquiums: 18.11.2004

Dekan: Prof. Dr. Jörg Eschmeier, Universität des Saarlandes

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfmittel angefertigt habe. Die aus anderen Quellen oder indirekt übernommenen Daten und Konzepte sind unter Angabe der Quelle gekennzeichnet.
Die Arbeit wurde bisher wieder im In- noch im Ausland in gleicher oder ähnlicher Form in einem Verfahren zur Erlangung eines akademischen Grades vorgelegt.

Saarbrücken, den ____________________

____

Short Abstract

Failure of computer software in a hard real-time system leads to severe consequences and must be avoided by proving the correctness of the system's software. A prerequisite for this is the determination of an upper bound for the worst-case execution times (WCET) of the tasks in the system. We show that for modern CPUs, WCETs can be obtained by static program analysis methods even for CPUs with execution history sensitives components like caches and pipelines. This is the first time that complex CPU features (out-of-order execution, speculation, etc) have been included in a comprehensive and safe analysis.

The approach presented in this thesis is able to handle the analysis of very complex architectures (PowerPC 755) by first modeling the CPU and peripherals of the system and then using abstractions on some components of the system to obtain an analysis. The analysis computes WCET for the basic blocks of the program by simulating the abstract system model. The correctness of the approach is shown.

A tool has been built based on this approach, which was evaluated under real-life industry conditions by Airbus France in the course of the DAEDALUS project, showing the practical applicability of the methodology.

Kurze Zusammenfassung

Fehlverhalten der Computersoftware eines harten Echtzeitsystems kann katastrophale Folgen haben. Um ein solches Verhalten zu verhindern, muss die Korrektheit der Programme des Systems vorher nachgewiesen werden. Eine Voraussetzung hierfür ist die Kenntniss von oberen Schranken für die Ausführungszeit der Programme (WCET). Für moderne CPUs können solche Schranken effektiv nur durch statische Analysemethoden verlässlich gewonnen werden, da die Laufzeiten stark von kontextsensitiven Komponenten (Caches, Pipelines) abhängen. Bisher galten komplexe Merkmale moderner CPUs (out-of-order Ausführung, Spekulation) als nicht effizient statisch analysierbar.

Die vorliegende Arbeit präsentiert einen Ansatz, der in der Lage ist, sehr komplexe Architekturen (etwa den PowerPC 755) zu behandeln. Hierbei wird zuerst ein Modell des Prozessors und der Peripherie des Systems erstellt, dessen Komponenten dann geeignet abstrahiert werden können, um eine Analyse zu erhalten. Die Analyse berechnet WCET für die Basisblöcke eines Programmes durch Simulation des abstrahierten Prozessormodells. Die Korrektheit der Analyse wird durch die Verwendung der Theorie der abstrakten Interpretation garantiert.

Mit diesem Ansatz wurde ein Werkzeug entwickelt, welches unter Industriebedingungen von Airbus France im Verlauf des DAEDALUS Projektes evaluiert wurde. Dabei konnte die praktische Anwendbarkeit des vorgestellten Ansatzes klar demonstriert werden.

Abstract

Hard real-time systems are computer systems that control critical physical plants (avionics, automotive, nuclear power plant control, weapon guidance, etc). If such a computer system fails, the consequences can be severe: damaged property or even loss of lifes. Therefore, hard real-time systems must be checked for correctness before being deployed. One aspect of correctness is the *timely response* of the system, often expressed by temporal deadlines that must be met by the software tasks in the system. An essential component for proving that every task meets its deadline is the knowledge of an upper bound for the execution time of each task, then *worst-case execution time* (WCET). It has been shown to be usually practically impossible to obtain the WCET by *measuring* real execution times of tasks, due to the complex dependencies between the execution time and the input data or starting conditions of the system. Thus, safe upper bounds for the WCET can only be obtained by *statically analyzing* a task's program code. Due to the restricted forms of programming used in hard real-time systems, it is in principle possible to compute a WCET from the program text alone (loop iteration and recursion bounds are assumed to be known).

Modern CPUs use features like caches and pipelines to improve performance. These features can lead to a huge variation in execution time as they are history sensitive: a memory access, e. g., may take just one cycle for an access that hits in the cache, while it may take more than 50 cycles for a cache miss and subsequent access to main memory. Whether an access hits in the cache depends on the contents of the cache and thus on the accesses performed before that access. A similar observation holds for effects in the processor's pipeline. The static analysis of such features in todays CPUs has proven difficult. Until recently, the static analysis of CPUs featuring branch prediction, out-of-order execution or speculation has been viewed as too complex to be used in practice [Eng02].

We present a novel approach that is able to analyze architectures with the before mentioned features for real-life sized example programs. Our approach is based on a *model* of the CPU and the peripherals (memory, system controller,. . .). We use a cycle-precise model with communicating units which have inner state and update rules. This resembles approaches taken by hardware description languages like VHDL or Verilog. The framework of *abstract interpretation* is used to define (safe) abstractions for some of the components of the model (e. g. the caches), reducing the model to details relevant to timing and to a size that can be practically handled. If the abstractions performed satisfy certain conditions, following from the theory of abstract interpretation, they are guaranteed to be safe in the sense that every concrete model state is subsumed by an abstract one. The *abstract model* obtained by this process can still be simulated cycle-wise and its simulation gives safe WCETs of the basic blocks of a program, while taking

into account pipeline effects even across basic block boundaries. The WCETs for the basic blocks together with the control-flow graph of the program can then be transformed into an integer linear program with the global execution time as the objective function to be maximized. The solution of this ILP is then a safe WCET of the program. This work presents two models in detail, namely for the Motorola ColdFire 5307 and the PowerPC 755, together with the abstractions used to obtain the abstract model for both processors. The abstract model simulation is then used as the transfer function of a data-flow analysis (DFA) over the control-flow graph of the binary program to be analyzed. This DFA is then the core of the implementation of the WCET analysis for the program.

This approach has led to the development of a commercial tool, **aiT**, which is based on provably correct methods and able to handle complex architectures for the computation of WCETs for real-life sized programs. Its prototypes have been evaluated during the DAEDALUS project by Airbus France with realistic benchmarks for avionics software under industry conditions. It has been awarded an European IST award 2004. This demonstrates the practical relevance and applicability of the approach presented in this thesis.

Other aspects that are of relevance for the real-life utilization of WCET tools are discussed in this thesis: validation of WCET tools and predictable hardware. Validating the results of a WCET tool is critical if it is to be deployed in a critical area like avionics. As our tool is based on provably correct abstractions, validation serves to detect implementation errors or errors made in the model itself. Predictable hardware means hardware whose worst-case behavior does not differ too much from its average-case behavior, making WCET prediction easier and more precise. The identification of problematic features in processors that have a bad worst-case behavior is thus an important step for design decisions in future real-time systems.

The approach can be extended in several ways, by using different modeling techniques, e. g. a model obtained from the authoritative VHDL code of a processor by abstraction steps. This approach is being studied in the AVACS Transregional Collaborative Research Center 14 sponsored by the German Research Foundation DFG. More processors (Motorola PPC 5xx, Infineon Tricore, Ti TMS320C33, Motorola STAR12) have been or are being modeled in this framework at the moment by the company AbsInt.

Zusammenfassung

Als *harte Echtzeitsyteme* bezeichnet man allgemein Computersysteme, welche kritische physikalische Umgebungen kontrollieren (etwa Flugzeugsteuerungen, Atomkraftwerkssteuerungen, Waffenlenksysteme, etc). Falls ein solches Computersystem Fehler aufweist, können die Konsequenzen drastisch sein: Hohe Sachschäden oder sogar Verlust von Menschenleben. Daher müssen harte Echtzeitsysteme vor dem Einsatz auf ihre Korrektheit überprüft werden. Ein Aspekt der Korrektheit ist die *rechtzeitige Antwort* des Systems, die oft durch Zeitschranken angegeben wird, die von den Tasks im Computersystem eingehalten werden müssen. Ein wesentlicher Bestandteil des Nachweises, dass jede Task ihre Zeitschranke einhält ist die Kenntniss der Ausführungszeit der Task im schlimmsten Fall, ihrer *worst-case execution time* (WCET). Es hat sich meist als nicht praktikabel heraus gestellt, die WCET durch *Messen* realer Ausführungszeiten zu bestimmen, da komplexe Abhängigkeiten zwischen der Ausführungszeit und den Eingabedaten oder Startzuständen des Systems bestehen. Daher können sichere obere Schranken für die WCET nur durch Benutzung *statischer Analysemethoden* auf dem Programmcode erhalten werden. Da die Formen der in harten Echtzeitsystemen benutzten Programmierstile eingeschränkt sind (keine Benutzung von Zeigern, kein Heap, beschränkte Rekursion, etc), ist es prinzipiell möglich, eine WCET allein vom Programmcode her zu berechnen (bekannte Schleifendurchlauf- und Rekursionsschranken).

Moderne CPUs benutzen Verfahren, etwa Caches und Pipelines, zur Erhöhung ihrer Performanz. Diese Verfahren können zu einer hohen Varianz der Ausführungszeit führen, da sie von der Ausführungsgeschichte abhängen: Ein Speicherzugriff z.B. kann lediglich einen Takt in Anspruch nehmen, wenn er ein Cachetreffer ist, oder er kann über 50 Takte dauern, wenn das gewünschte Datum nicht im Cache liegt und aus dem Hauptspeicher geholt werden muss. Ob ein Zugriff ein Cachetreffer ist hängt von dem Cacheinhalt und daher von den vor diesem Zugriff ausgeführten Zugriffen ab. Ähnliches gilt für die Effekte innerhalb der Prozessorpipeline. Die statische Analyse solcher Eigenschaften heutiger CPUs hat sich als schwierig erwiesen. Bis vor kurzem wurde die Analyse von CPUs mit Sprungvorhersage, out-of-order Ausführung oder Spekulation als zu komplex für den praktischen Einsatz angesehen, vgl. [Eng02].

Wir stellen einen neuen Ansatz vor, der in der Lage ist, Architekturen mit den oben genannten Eigenschaften für realistische Programme zu analysieren. Unser Ansatz basiert auf einem *Modell* der CPU und der Peripherie (Speicher, System Controller). Wir benutzen ein zyklengenaues Modell mit untereinander kommuniziernden Einheiten, welche einen inneren Zustand aufweisen und mit Zustandsübergangsregeln ausgestattet sind. Dies lehnt sich an Ansätze aus dem Gebiet der Hardwarebeschreibungssprachen an, etwa VHDL oder Verilog. Im An-

schluss an die Modellierung werden mit Hilfe der Theorie der *abstrakten Interpretation* (sichere) Abstraktionen für einige der Komponenten des Modells definiert, die das Modell auf die für das Zeitverhalten wichtige Bestandteile reduzieren und die Grösse des Modells auf ein handhabbares Ma"s bringen. Wenn die durchgeführten Abstraktionen gewisse, aus der Theorie der abstrakten Interpretation stammende, Bedingungen erfüllen, ist garantiert, dass jeder konkrete Modellzustand von einem abstrakten Zustand repräsentiert wird, d.h. die Abstraktionen sind sicher. Das durch diesen Prozess erhaltene *abstrakte Modell* kann immer noch zyklenweise simuliert werden, wodurch man sichere obere Schranken für die Ausführungszeit der Basisblöcke des Programms bekommt. Pipelineeffekte werden zusätzlich über Basisblockgrenzen hinweg propagiert. Die WCETs für die Basisblöcke, zusammen mit dem Kontrollflussgraphen des Programmes können dann in ein ganzzahlig-lineares Programm überführt werden, wobei die Gesamtausführungzeit des Programmes als zu maximierende Zielfunktion formuliert wird. Eine Lösung dieses Programmes ist dann eine sichere obere Schranke für die Ausführungszeit des Programmes. Diese Arbeit präsentiert zwei detaillierte Modelle für den Motorola ColdFire 5307 und den Motorola PowerPC 755 zusammen mit den zur Erlangung des abstrakten Modells durchgeführten Abstraktionen. Die abstrakte Simulation wird dann als Transferfunktion einer Datenflussanalyse (DFA) über dem Kontrollflussgraphen des Programmes benutzt. Diese DFA ist dann der Kern der Implementierung der WCET Analyse des Programmes.

Dieser Ansatz hat zur Entwicklung eines kommerziellen Werkzeuges, **aiT**, geführt, welches auf beweisbar korrekten Grundlagen basiert und in der Lage ist, komplexe Architekturen zur Berechnung von WCETs realistischer Programme zu behandeln. Seine Prototypen wurden während des DAEDALUS Projektes durch Airbus France mit realistischen Benchmarks für Avionics Software unter industriellen Bedingungen evaluiert. Das Werkzeug erhielt einen IST Preis der Europäischen Union. Dies zeigt die praktische Relevanz und Anwendbarkeit des vorgestellten Ansatzes.

Andere, für die reale Anwendbarkeit von WCET Werkzeugen wichtige Aspekte werden ebenfalls diskutiert: Validierung von WCET Werkzeugen und Vorhersagbarkeit von Hardware. Die Validierung der von WCET Werkzeugen gelieferten Ergebnisse ist wichtig, wenn sie in einem Bereich wie der Avionic eingesetzt werden sollen. Da unser Werkzeug auf beweisbar korrekten Abstraktionen beruht, dient die Validierung hier nur der Absicherung gegen Implementierungsfehler und Fehlern im Modell. Vorhersagbare Hardware bezeichnet Hardware, deren Verhalten im schlimmsten Fall nicht weit entfernt ist vom Verhalten im durchschnittlichen Fall. Dies macht die Vorhersage von WCETs einfacher und präziser, da weniger "Unfälle" im Verhalten der Hardware modelliert oder konservativ abgeschätzt werden müssen. Die Identifikation problematischer Eigenschaften in Prozessoren, welche ein ungünstiges Verhalten im schlimmsten Fall ausweisen, ist daher ein wichtiger Schritt im Entwurf zukünftiger Echtzeitsysteme.

Der vorgestellte Ansatz kann in verschiedene Richtungen durch die Benutzung andere Modellierungstechniken erweitert werden. Z.B. könnte ein VHDL Modell eines Prozessors durch Abstraktionsschritte auf ein abstraktes Modell reduziert werden. Diese Erweiterung wird zur Zeit im transregionalen Sonderforschungsbereich AVACS der DFG untersucht. Derzeit werden zusätzliche Prozessoren (Motorola PPC 5xx, Infineon Tricore, Ti TMS320C33, Motorola STAR12) im Rahmen dieses Ansatzes von der Firma AbsInt modelliert.

Acknowledgements

Many people have contributed in different ways to this work.

First, I thank Prof. Wilhelm for letting me work on this interesting subject; he gave the direction for my work and contributed his vast knowledge and advice in many discussions that shaped the style and contents of this thesis. He created a unique working atmosphere in his group, making the cooperative work with all the members of the group very delightful.

Then, my colleagues deserve a big thanks for their discussions and suggestions regarding this thesis. Especially Marc Langenbach, Sebastian Winkel, Christian Probst, Jörg Bauer, Daniel Kästner and Stephan Diehl did a great job at commenting on drafts of this thesis.

The people at AbsInt Angewandte Informatik have contributed in many different ways to this work. Reinhold Heckmann was an invaluable help in looking at things the right way, asking the right questions and commenting on drafts of my work. Christian Ferdinand and Florian Martin provided great insights and implementation contributions. Most critical parts of the WCET framework were implemented by Florian Martin, Michael Schmidt and Henrik Theiling. Without this framework and the work done at AbsInt the results of this thesis would not have been achieved. Also, Reinhold Heckmann and Christian Ferdinand helped me very much by reviewing drafts of my thesis, pointing out the errors and missing ends.

The work described in this thesis has been done in the DAEDALUS project supported by the European FP5 program under RTD project IST-1999-20527.

In the DAEDALUS project, the people from Airbus France, Jean Souyris and Famantanantsoa Randimbivololona provided a remarkable amount of practical experiences and interest in the results of our WCET work. Without their help, this work would never have reached its present level of practical usefulness.

My family has always been a big support for me. I am grateful to my parents, Helga and Josef, for helping me and waiting patiently for their son to finally finish his PhD work. I hope it was worthwhile to wait for this thesis:-)
항상 저를 자상하게 지켜봐 주시고, 후원해 주신 어머님과 아버님께도 이 논문을 통해 깊은 감사를 드립니다.

Finally, I have to thank my son Raphael 찬준 for being the child he is. My wife Yoomi deserves my highest gratitude for helping me and taking care of our family while I was busy filling the pages of this thesis. Her asian patience was really put to the test sometimes.

For my wife Yoomi.

Contents

Chapter 1

Introduction

This thesis is concerned with finding tight and safe predictions for the maximal execution time of programs (also called *worst-case execution time* or *WCET* for short). Finding upper bounds is important to guarantee that the response of a computer system is always computed in time and that the system can thus react to external events in a guaranteed time frame (a system that has such temporal correctness requirements is called a *real-time system*). Finding a *tight* upper bound is important in order to have a well scaled system, not reserving system power for computing needs that will never arise in practice.

Although this problem is undecidable in general, it is decidable for a restricted set of programs that satisfy the following constraints: static bounds to recursion and to loop iterations must be known. For this restricted type of programs, the problem seems to be easily solved, by applying, e. g., timing schemes to the program structure ([Sha89]). However, mostly these methods are only defined for high-level programming languages or make certain assumptions about the locality of timing behavior.

Another aspect of the work presented in this thesis was practical applicability. While some methods have been presented to compute WCET (bounds), they ignored the fact that certain conditions are imposed on the usage of a tool derived from these methods. In the course of the DAEDALUS[1] project, a tool based on the principles presented in this thesis has been implemented and evaluated by industrial end users. The feedback gained through this process has lead to an efficient and realistic tool, called **aiT**[2], which is sold by the company AbsInt[3]. This tool has been awarded an European IST Prize for 2004.

In the remainder of this chapter, the topics of real-time systems and WCET

[1]See `http://www.di.ens.fr/~cousot/projects/DAEDALUS`. The project has been supported by the EU under IST-1999-20527.

[2]`http://www.AbsInt.com/ait`

[3]`http://www.AbsInt.com`

determination are introduced and our approach to the latter is sketched. The problems for WCET determination introduced by modern hardware are sketched in Section 1.3. Related work is discussed in the last section of this chapter.

This thesis tries to collect all the material necessary for a full understanding of the issues involved in the analysis of modern hardware for WCET prediction. We deem this necessary since the application of an industry strength WCET prediction tool requires a lot of work in the framework surrounding the tool. On the other hand, the problems that occur for a safe design and implementation of such a tool necessitates a deeper reflection about the hardware and software components that make such a task complex and demanding.

As a consequence, many concepts are treated with considerable verbosity and a lot of material from other sources is included. Therefore, we summarize the new contributions of this thesis for clarity here. A technique to describe the hardware that is to be analyzed has been developed: pipeline models. Clear semantics and examples of applications have been given: Motorola MCF 5307 and Motorola PowerPC 755. An analysis and its correctness are presented: pipeline analysis. A discussion of hardware features that make precise predictions difficult is presented: Chapter 8. The problem of the verification of an implementation of an analysis is discussed and solutions are suggested: Chapter 7. The real-life readiness of the resulting analysis is shown and problems in the implementation of the analysis are discussed.

The other chapters of this thesis are organized as follows: in Chapter 2, hardware features of modern processors which make computing the WCET difficult are introduced. Two processors are presented in detail, the Motorola ColdFire 5307 and the Motorola PowerPC 755. For both processors, a WCET tool has been implemented. In Chapter 3, the theoretical framework for the analyses behind our WCET tool is presented, i. e. *abstract interpretation*. Chapter 4 introduces the method used to model the timing behavior of processors, and presents the models for the MCF 5307 and the PPC 755. The analyses based on these models are presented in Chapter 5 together with the necessary correctness proofs. The tool using these analyses and the remaining framework needed for performing WCET analyses on machine programs are presented in Chapter 6, while the important question on how to verify the correctness of the implementation of the tool is discussed in Chapter 7. General comments and suggestions for the design of predictable (w.r.t. WCET determination) systems are collected in Chapter 8. Chapter 9 summarizes the contributions of this thesis and gives directions for future work.

1.1 Real-Time Systems

Embedded systems are computer systems interacting with a physical environment, e. g. flight-control computers in planes, DVD players, nuclear power plant control, etc. Using computer systems to control or interact with a physical environment adds another level of complexity to the system. Not only must the system be functionally correct, i. e. compute the correct result from its inputs, but also the time it takes until the result is available is important. This is because the results are usually used to control actuators that influence the physical environment, leading to new system parameters, which are again inputs for the computer system. If the results are available too late, the computed action may be inappropriate for the correct functioning of the whole system. As an example a new angle for the rudder of a plane that is applied too late may cause a resonance and oscillation of the plane, leading to a crash. Or, less dramatically, an audio frame delivered too late while playing a movie causes audio disturbance and loss of synchronization between picture and sound.

Systems for which timely functioning is important are called *real-time systems*. Timing constraints are often formulated as *deadlines*, i. e. points in time until which all results must be available. If missing timing constraints only occasionally is acceptable and does not cause the whole system to malfunction, the system is a *soft real-time system*. Systems that will not work properly if timeliness is not fulfilled are called *hard real-time systems*. In a hard real-time system missing a deadline can lead to a loss of life and/or property in the physical environment controlled by the system. Examples of hard real-time systems are avionics, especially flight-control systems, missile guidance, nuclear power plant control or steering controls in cars. This thesis is primarily concerned with analyzing hard real-time systems, with the focus on avionics. However, the results can be fully applied to any hard real-time system and at least partially to soft real-time systems as well.

For our setting, we look at real-time systems that are made up of a set of *tasks*. Tasks are executed periodically and implement a certain sub-function of the system. There may be dependencies between tasks T_i and T_j such that T_i must finish its execution before T_j starts executing. We assume that these dependencies are given by a partial order $\leq$ on the set $\{T_i \mid 1 \leq i \leq n\}$ of tasks. Also, there are parameters associated with each task T_i: a task may only be started after a certain point of time, that is there is a minimal *release time*, r_i for it. The *deadline* of the task is denoted by d_i and the task's *ending time* by e_i, the starting time by s_i. We only consider uniprocessor systems, so all the tasks of the system must be executed on one CPU. As there are normally several tasks without dependencies ready to be executed at the same time, we introduce a *priority* among tasks, a total order $<$ on the set of tasks. The task with the highest priority is the smallest one in this ordering. From the tasks that are *ready* (i. e. have no dependencies) the *scheduler* selects one to execute on the CPU. This *scheduling* can be either *static* or *dynamic*, depending on when the decisions for task selection are made: offline before the system starts to execute or online by scheduling code. Furthermore, another characteristics of a real-time system is, whether the scheduling is *preemptive* or not. With preemptive scheduling, the currently executing task is preempted from execution and the next highest priority task is executed as soon as it becomes ready (if, e. g., its release time has passed). With nonpreemptive scheduling, the currently executing task is always executed until it finishes. Real-time systems are designed by fixing certain *periods* of execution of the subtasks in the systems (coupled to sampling rates of sensors, etc). Thus, also the tasks are assigned an individual period p_i. Two invocations of a task are separated by this period.

There are several strategies how the priority ordering $<$ on tasks is computed. The ordering can be static, giving fixed priorities to tasks, or dynamic, where priorities are determined at run-time based on certain conditions. A static one is, e. g., *rate monotonic*, [LL73], where the priorities are assigned by inverse length of the period, i. e. the task with the smallest period is assigned the highest priority. A dynamic strategy is *earliest deadline first* (EDF), where the highest priority is assigned to the task whose deadline is nearest. EDF with preemptive scheduling is known to be optimal on a uniprocessor system, cf. [Liu00]. A schedule obtained by applying a given scheduling strategy is *feasible*, if every task meets its deadline, i. e. $\forall i : s_i + e_i \leq d_i$ and the dependencies among the tasks and earliest release times are met.

For all strategies there exist sufficient criteria that guarantee that a feasible schedule can be generated if they are fulfilled, cf. [Liu00] or [BW90]. Such a criterion is also called *schedulability test*. E.g., in the *response time approach*, [JP86, TBW94], the maximal time, R_i, from the release of a task until it finishes its execution is described by a recursive formula which takes into account the preemptions of the task by other tasks with higher priorities and because of dependencies. In the simplest form the equation is (tasks T_j with lower indices j have higher priority):

$$R_i^{(n+1)} = e_i + \sum_{k=1}^{i-1} \left\lceil \frac{R_i^{(n)}}{p_k} \right\rceil e_k$$

The start value can be chosen to be $R_i^{(0)} = e_i$. The recursive computation of the $R_i^{(j+1)}$ can be stopped if $R_i^{(j+1)} = R_i^{(j)} (=: R_i)$ for some j or if $R_i^{(j)} > d_i$ in which case no feasible schedule exists. The necessary schedulability condition is then $\forall i : R_i \leq d_i$.

All these criteria need the WCETs of the tasks as inputs. More precisely, they need *upper bounds* for all the WCETs of the tasks. In the literature, the notions

"WCET" and "upper bound for the WCET" are usually not distinguished. So, in the following, when we talk about WCET we actually mean an upper bound for the WCET. Thus, knowledge about the WCET e_i of each task T_i is crucial for proving that a feasible schedule for the whole real-time system exists so that it can function correctly.

The remainder of this thesis concerns itself with obtaining a tight upper bound for the WCET of a task. The *best case execution time* (BCET) of a task is sometimes also of importance to get a feeling for the utilization of the processor or for multiprocessor scheduling. Mostly, the execution time of a task will be less than its WCET. In the interval from the end of the task until its WCET has passed, additional soft real-time tasks may be executed without the possibility of violating the feasibility of scheduling (slack time). The maximal available slack time can be bounded by the BCET of the tasks. The method presented later to compute (upper bounds to) WCET can equally well be utilized to compute (lower bounds to) BCET of tasks. In fact, all that is needed is redefining the ordering $\sqsubseteq_\times$ of the abstract domain in Section 5.1 to use $\geq$ to sort the second component of each pair instead of $\leq$, adjusting the least upper bound and greatest lower bound operators as well as redefining γ as

$$\gamma(\check{s},\hat{m}) = \bigcup\{(s,m) \mid s \in \Gamma(\hat{s}) \wedge \hat{m} \leq m \leq T_{\max} \wedge \hat{s} \in \check{s}\}$$

1.2 WCET Analysis

In general it is not possible to obtain running times for programs due to the halting problem. But for real-time systems it is possible to only use a restricted form of programming which guarantees that programs always terminate. That is, recursion is not allowed (or restricted) and the iteration counts of loops are known in advance, or at least lower and (finite) upper bounds for them are known.

Two classes of methods to obtain WCETs can be distinguished:

- *Dynamic* methods utilize real program executions to obtain WCETs
- *Static* methods only need the program itself, maybe extended with some additional information (like loop bounds)

1.2.1 Dynamic Methods

The easiest dynamic method is *measuring* the execution times of a number of real program executions, either by augmenting the program with additional code to manipulate timer hardware or by using a logic analyzer to observe the relevant signals directly at hardware level (e. g. a write by the program to a special memory location indicating start and another one indicating end of execution). This method has two drawbacks: it is not *safe* and in practice it involves a lot of effort in terms of measuring equipment and input simulation. It is not safe, because in virtually all cases the worst case *input* to a program[4] is not known and thus the WCET cannot directly be measured. Thus, it can only be guaranteed that the WCET has been obtained by measuring for *all* inputs, which is practically infeasible. The method is very expensive in terms of resources needed (both hardware and personnel) as the computer system controlling a complete real-time system is connected by sensors and actuators to its environment. For the measurements, sensor data has to be provided by other external simulation hardware, since these measurements cannot be done in the real system for safety reasons. Also, additional hardware has to compute the effects of the actuator outputs and feed them back to the sensors. This environment is needed to make sure that the program run behaves as in the real system. If the sensor inputs in the measurement run differ from those in the real system, the program may also behave differently, e. g. other program paths are taken based on the input range of a sensor.

Additional drawbacks of this approach are that some variants need to use modified code (augmented with instructions to steer timers) or utilize a two-loop approach: a loop is put around the piece of code under examination. The execution time of several loop iterations is measured to increase reliability and precision of the results. This is compared against a measurement of the loop execution with an empty body to separate the execution time of the loop instructions from the execution time of the investigated code snippet's. However, a certification requirement makes this method difficult: validation of real-time systems, especially in avionics, mandates that validation is done on *exactly the same* code that will fly in the airplane later[5]. Using instrumented code is not possible under these restrictions.

Another method combines measuring and static methods. Here, small snippets of code are measured for their execution time, then a *safety margin* is added and the results for code pieces are combined according to the structure of the whole task. E.g. if a task first executes a snippet A and then a snippet B, the resulting time is the one measured for A, t_A, added to the time t_B measured for B: $t = t_A + t_B$. This reduces the amount of measurements that have to be made, as code snippets tend to be reused a lot in control software and only the different snippets need to be measured. However, it causes the need to argument about the correctness of the composition step of the measured snippet times. Correctness depends, e. g. on certain properties of the measurement environment of the snippets. One example would be that the snippets are measured with an empty cache at the beginning

[4]I.e. the input leading to the WCET.

[5]Test what you fly and fly what you test.

under the assumption that this will lead to a larger execution time than with a partially filled cache. In Section 2.5.5 we will show that especially this assumption can be wrong. The problem of unknown worst-case input exists for this method as well, while it is still infeasible to measure for all input values.

A practical disadvantage of these dynamic methods is that the hardware (and the surrounding simulation equipment) must be available. In early stages of development this is not the case. Design decisions about the layout of the system must be made before results about worst-case performance are available. Since it is extremely costly to change designs in later stages of the development process, this puts a high responsibility and risk on the engineers building the system.

Program		Timing
`If`	`(a<b) Then`	1
	`c:=a+b;`	2
	`d:=1;`	1
`Else`		1
	`d:=0;`	1
`Fi`		4

Figure 1.1: Simple program with execution times

1.2.2 Static Methods

The second class of methods does not rely on executing code on real hardware but rather takes the program code itself, combines it with some (theoretical) model of the system and obtains WCETs from this combination. A simple variant, applicable to a wide range of programming languages and systems, is the use of *timing schemes*, [Sha89]. Here, execution times are assumed to be known for the atomic instructions of the program and the execution time for the whole program is obtained by combing results for parts of the program. E.g., for the program in Figure 1.1, the execution time of the whole program is made up of the time to evaluate the condition of the `If` statement plus the maximum of the execution times of the two branches of the `If` statement, after the `Then` and the `Else` keyword. In the `Then` branch, the execution time of the two consecutive instructions is the sum of both instructions, in this case 3. For loops, iteration bounds are used.

For WCET computation, the maximal iteration count is multiplied with the time for the body plus the time for the evaluation of the loop's condition.

This method gives a safe approximation to the WCET, if all the atomic execution times are WCETs and the upper loop bounds are correct (i. e. every loop iteration count during program execution is smaller than the given bound). Precision can be very bad, if e. g. a loop iterates 100 times, but the WCET of the body, e_{body} only really occurs during one of these iterations and the others are considerably faster (say twice as fast). The overapproximation is $99 * 0.5 * e_{\text{body}}$. Another source of imprecision are the loop bounds: if they are overly conservative, the result is still safe but far away from the real WCET.

A different source of problems arises from some implicit assumptions underlying this method. Timing schemes assume that local worst-case execution times are *invariant*. This means that the WCET obtained for a part of the program, say one instruction, does not change if the same part later occurs again in the program. E.g., in Figure 1.1, if the `d:=0;` instruction would occur again in another part of the program, it still would take 1 time unit to execute, not two or three. For loops, one assumption is the *monotonicity* of the loop body. This means that if one executes a loop *locally* more often, this will *globally* also lead to a higher execution time, so we can use the upper bound to the loop iterations as a safe number of iterations to compute WCET. Both assumptions are not valid for systems where execution times are dependent on the *execution history*. Unfortunately, modern processors with caches and pipelines are sensitive to the execution history in their execution time.

Related to timing schemes is the approach of modeling the flow of a program as an *integer linear program* (ILP) with local worst case execution times at atomic instructions. The WCET is computed by solving the ILP to maximize a formula defining the execution time, cf. [LMA97].

An example **C** program and the ILP variables associated with it are depicted in Figure 1.2. Here, the variables $\mathbf{a}, \mathbf{b}, \ldots$ on the right hand side are the execution counts for the corresponding instructions. The t_m are the execution times for the instructions and the x_i are the traversal counts of edges.

How these variables are connected is shown in Table 1.1. The inequality $0 \leq x_4 \leq 10$ comes from the loop iteration bound. With given x_i, the execution time of the whole program is $t = \sum_{n \in \{\mathbf{a}, \mathbf{b}, \ldots\}} n \cdot t_n$. To determine an upper bound to the WCET of the program, we solve the ILP by maximizing t, obtaining values for the x_i and $\mathbf{a}, \ldots$. Additional constraints can be added to the ILP to increase the precision of the solution. In the example, the constraint $0 \leq x_5 \leq 6$ can be added because instruction e is executed at most six times.

Another static method is to simulate the program in a model of the system, namely the processor. Such simulators for processors are made available by the

```
a:  j:=0;
b:  i:=0;
c:  while (i<10) {
d:    if (i<6)
e:      j+=i+1;
      else
f:      j++;
g:    i++;
    }
h:  j++;
```

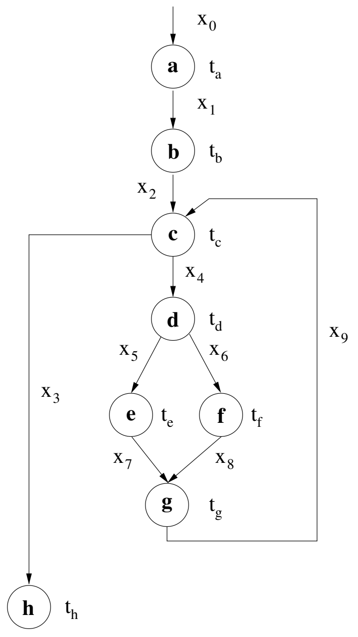


Figure 1.2: Example program with ILP variables

manufacturers of the processors to ease code development for embedded platforms, where the usual debug techniques cannot be applied. Unfortunately, simulation will be much slower than running in the real system. The problems of measuring, i. e. unknown worst-case input and the number of possible inputs apply to this method as well. They can partly be circumvented by using some form

$a = x_0$	$b = x_1$	$c = x_2 + x_9$	$d = x_4$
$e = x_5$	$f = x_6$	$g = x_7 + x_8$	$h = x_3$
$0 \le x_4 \le 10$	$x_5 + x_6 = x_4$	$x_9 = x_7 + x_8$	$x_7 = x_5$
$x_8 = x_6$	$x_0 = 1$	$x_1 = a$	

Table 1.1: Relations of example ILP variables

of abstraction for the simulation, cf. the discussion of Lundqvist's and Stenström's work [LS98, LS99] in Section 1.4.

It is also possible to use simulation only for small parts of the program, like basic blocks, and to compose the results to form the WCET of the whole program. If the small parts are considered in isolation, sufficient starting (i. e. worst-case) scenarios for the simulation must be used to achieve a sound analysis. This simulation is very sensitive to so called *timing anomalies* (see below), where local worst-cases may not lead to global worst-case scenarios but rather a locally better scenario results in the global worst-case. These effects must be considered in the worst-case scenarios for the simulation of the program parts by adding a safety margin. This forced localization of the analysis reduces computation time but can drastically decrease precision, if effects in the pipeline can reach over several basic blocks.

Model checking, [CGP99], is a very popular approach in the area of verification of embedded and real-time systems. In principle it can also be used to obtain WCETs for programs. For this, the system (processor and periphery) and the program itself must be coded as an automaton and an appropriate formula describing the execution of the program within a time t_1 must be constructed. Model checking can then determine if this formula is violated. If not, the WCET of the program is not larger than t_1. Otherwise, the procedure is repeated for the execution time $t_2 = 2 * t_1$, etc. In the same manner, a tighter WCET bound can be obtained after a successful check for t_1 by using a formula for $t_2 = t_1/2$, and rechecking, increasing the time again if the check fails, in analogy to binary search in sorted tables. It has already been argued in [Wil04] that this approach (and further variants, described in the paper) is inefficient for realistic systems if the program consists of low-level machine code. Obtaining the formula to check is also not trivial.

Abstract Interpretation of Pipeline Models

Finally, one can apply a method called *data flow analysis* (DFA) to the problem of determining WCETs. This method is well established in the area of compiler construction and used to obtain information about a program that holds for *all* executions of the program. It is, e. g., used to determine the values of variables as a prerequisite to constant folding and expression evaluation optimization. The charm of DFA is that it computes safety properties for each instruction of the program that are valid for all executions of the program. Since DFA allows to define *abstractions* of program values, like e. g. inputs, results are computed once over these abstractions. A well-founded theory, the framework of *abstract interpretation*, underlies this approach, making correctness proofs for the results obtained easier. This method also does not suffer from the drawbacks of measuring and simulation, namely unknown worst-case inputs and prohibitively large input value sets. However, it can be costly for large programs. Our approach presented in this thesis uses DFA to obtain WCETs for local parts of the program, namely basic blocks. The processor's execution is abstractly simulated, and ILP methods are utilized to combine these local worst cases into a global WCET. Unlike other approaches, our approach considers the global flow of the program also in the first DFA phase by propagating simulation results globally between basic blocks. Thus, worst-case assumptions need not be made by our analysis on basic block level. The DFA uses a set of abstract pipeline states as the domain for the values propagated through the CFG.

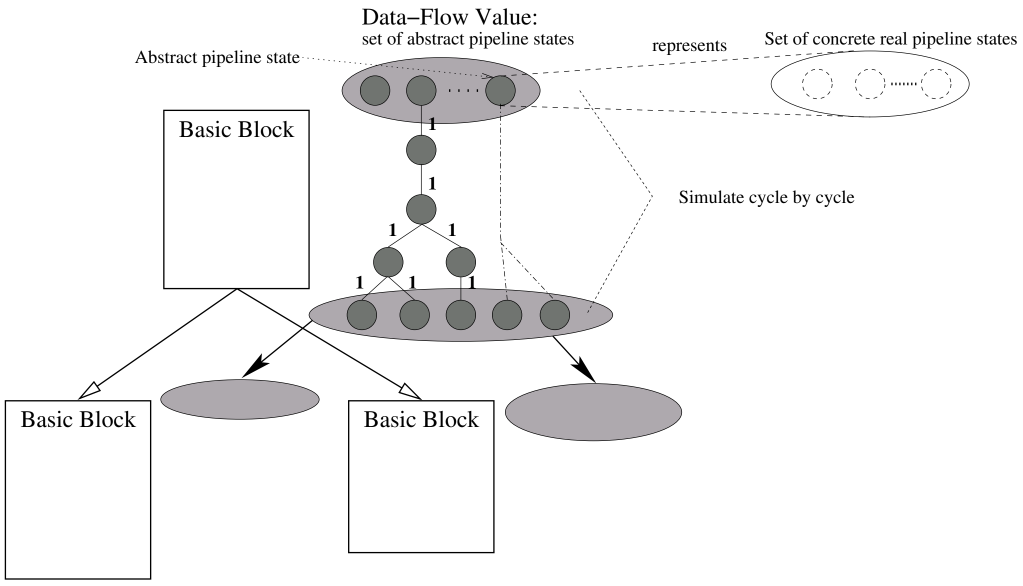


Figure 1.3: Overview of pipeline analysis by abstract simulation

As sketched in Figure 1.3, each abstract pipeline state describes a *set* of concrete pipeline states. A concrete pipeline state represents the complete system state relevant for timing, like cache contents, etc. A model defines the execution of the whole system by evolving concrete pipeline states clock cycle by clock cycle.

In the analysis, each abstract pipeline state is abstractly simulated cycle-wise. It is possible that more than one result state is generated because of information loss due to the abstraction that has been made. The maximal number of simulation cycles for the instructions in the basic block starting with all abstract states is an upper bound for the WCET of the basic block. Certain connections between the abstract and concrete states guarantee correctness of the abstract simulation on the abstract pipeline states. This approach allows to precisely model the system and still obtain efficient analyzers. In addition, the system hardware need not be available for the analyzer to deliver results. Furthermore, parameters of the model, e. g. the size of the cache or memory layout can be changed before each analysis to gain hints on system performance and to support design decisions. As the next section shows, modern hardware poses a lot of problems for WCET analysis by exhibiting non-local effects, because behavior is history dependent. Our approach allows to capture these effects in the model so that the analysis will correctly predict and incorporate them. The central part of the analysis, the system model and the abstraction over it, are described in Chapters 4 and 5, while the other components of the industrial tool built from this approach are described in Chapter 6.

1.3 Modern Hardware

As stated in the previous section, most WCET methods are not applicable if the instruction times of atomic components are not fixed. Modern processors have caches and pipelines to bridge the gap between slow memory access times and fast processor cores and to accelerate program execution by overlapping instruction execution. Caches and pipelines increase the performance of program execution, at least in the average case. However, they introduce a dependency on the execution *history* for an instruction's execution time. E.g., if a memory read accesses data already in the cache, the instruction will execute fast. If the instruction has to access main memory to fetch the data, the execution will take much longer. Clearly, whether the data is in the cache or not, depends on the accesses happening before this access. In order for this instruction to be a cache hit, there must have been an access putting the data into the cache. Furthermore, there must not have been too many other accesses to the cache because otherwise the loaded data may have been replaced by a different data block. Due to the huge number of cache configurations and of paths that lead to a single instruction, it is normally not possible to precisely predict the cache contents that occur when execution reaches that instruction.

The same holds, although less dramatically, for the behavior of a processor's pipeline. The execution time of an instruction depends on the instructions before it in the pipeline (and partly also on the instructions after it, due to out-of-order and speculative execution). The content of the pipeline at the execution start of an instruction clearly depends on how this instruction was reached during program execution, i. e. on the execution history.

Thus, one can no longer just look up the execution time of a single machine instruction in the processor handbook. The same effect makes measuring execution times difficult, as the dependencies on the execution history are not linear but can show chaotic behavior. That is, a small change in the value of an input parameter can lead to a huge change in the execution time of the whole program, as the cache and pipeline contents are changed and may suddenly cause a completely different behavior of the rest of the program. One example: the small change in an input value causes one special program path to be traversed. Fetching this program fragment causes other memory blocks in the cache to age so that they are in the sequel thrown out of the cache, while they stayed in the cache when another path was taken. Later, these blocks cause additional cache misses. This might increase the execution time by a factor of 30.

Clearly, this behavior makes is impossible to practically apply a technique like timing schemes from the previous section in order to obtain tight WCET bounds. One might be able to find a penalty for all the effects that may cause a certain instruction to prolong its execution, thus removing the history dependency of the instruction (by adding the penalty to the WCET of the instruction). But this penalty will clearly only occur for a few real executions. By using it for all instruction occurrences, the WCET obtained is too imprecise, rendering it useless. On the practical side, it has proved difficult to actually *find* a bounding penalty for effects on an instruction due to the complicated interactions between different processor features that may have effects on instruction execution. These interactions may even lead to *timing anomalies*, cf. Section 2.4, where locally faster executions (e. g. a cache hit) can lead to globally slower program execution.

The history dependencies also have other effects besides complicating WCET computation. As said before, the WCETs are needed for scheduling analysis to make sure that the tasks in the system can finish their execution before their deadlines. In systems that use preemptive scheduling, a task's execution may be interrupted by the scheduler and a higher priority task may be executed instead. The execution of the first task is resumed when the second task has finished its execution. At the point of the resumption of the task's execution, the history of the processor has changed compared to an undisturbed execution of the task: the cache contents may be different, the pipeline contents may be different, cf. [BN94]. This may cause an additional amount of execution time for the task to be required: e. g. because useful data has been replaced from the cache by the interrupting task and has to be reloaded from slow main memory.

The changed pipeline contents may cause strange alignment effects for program execution, leading to longer execution times (one example of such an effect is presented in Section 2.6.6, where the effect is only bounded by the program length, not by a processor dependent constant). Thus, the WCET of a task is no longer constant but becomes history dependent: it depends on which task has interrupted this task when and where. Most scheduling tests expect the WCET to be constant. Some extensions of schedulability tests have been presented, e. g. [Sch02] or [LHYM$^+$96]. However, these approaches are either too computationally complex or loose too much precision due to overly large overapproximations to possible damages. For systems with preemption, no practically feasible scheduling test exists today that can take history dependent effects of processors into account. For static scheduling, where a sequence of tasks (segments) is computed offline, a solution has been presented in [KT99] which takes the effects of caches and pipelines into account across switches to different segments of other tasks. A segment here is a linear piece of a task. The approach presented in the paper also allows to use precedences among and earliest release times for tasks.

Our approach only determines a WCET for the uninterrupted execution of a task. Effects caused by preemption have to be considered separately. Because we use an appropriately abstracted model of the processor and its periphery, we can capture all history dependent effects safely. As the results in Section 6.5 show, the results are also tight. In the next section we go on by discussing work related to our approach.

1.4 Related Work

Related work in two areas is discussed: determining WCET and modeling of processors or systems.

1.4.1 WCET determination

Based on Shaw's timing scheme [Sha89] which focuses on high-level language constructs and does not consider cache or pipeline effects, Lim et al. [LBJ$^+$95, LRM$^+$94] and Hur et al. [HBL$^+$95] present an analysis technique taking modern processor features into account. Cache effects are modeled via bookkeeping of first and last references to blocks, and reservation tables are used to handle pipeline effects. As the target machine –a MIPS 3000– implements only a simple pipeline, reservation tables whose resources are registers and pipeline stages are sufficient. For loops in the CFG, an approximation operator, related to a least upper bound operator in DFA, is used to determine cache contents and execution costs. Results were only reported for toy sample programs. More sophisticated processors featuring out-of-order execution, superscalarity, or set-associative caches are not considered. Correctness of the approach is not proven, nor is it clear how this could be done.

The approach was extended in [LHYM$^+$96] to incorporate the effects of preemption to the WCET computation of tasks. The approach uses the response time approach for fixed priority scheduling (rate monotonic) and integrates the preemption effects into the recursion equations for the response times in the form $r_i = e_i + \sum_{T_j \in \mathrm{hp}(i)} \left\lceil \frac{r_i}{p_j} \right\rceil e_j + \mathrm{PC}_i(r_i)$. Here, $\mathrm{hp}(i)$ is the set of tasks with higher priorities than T_i and $\mathrm{PC}_i(r_i)$ is the execution time penalty incurred on T_i. It is computed by solving an ILP that describes the maximal damage in terms of replaced cache contents of the tasks. The constraints of this ILP are computed from the useful cache blocks of a task. A useful cache block is a cache block that may be referenced again after it has been loaded. Solving this ILP for each iteration of the computation of r_i seems to be very complex and only very small examples are presented. Pipelines are not considered in the paper and it is not clear how timing anomalies should be integrated into the approach efficiently.

Healy et al. [HAM$^+$99, HWH95, MWH94] present another approach for predicting WCETs in the presence of caches and simple pipelines. In a first step of the analysis, a static cache simulator classifies instructions as cache hits or misses. This information is used by a pipeline path analysis that computes the execution time for a sequence of instructions. Loops are handled in a bottom-up manner. Only the simple pipeline of a MicroSPARC is considered and in [HAM$^+$99] only direct-mapped caches are taken into account. The method can only be applied to pipelines which can be described by resource usage patterns of instructions. For their experimental results, the authors only consider a small direct-mapped cache with small test programs. Complex architectures with timing anomalies are not considered and can in fact not be handled by simple resource usage patterns.

Li et al. suggested another solution using the technique of integer linear programming [LMA97]. Both cache and pipeline behavior prediction are formulated as one linear program. The Intel i960KB [Int91] is investigated which has a fairly simple pipeline. So only structural hazards need to be modeled keeping the complexity of the integer linear program moderate. Branch prediction and/or instruction prefetching are not considered at all. Obtaining the ILP modeling for a more complex processor will be difficult. Using their approach for super-scalar pipelines does not seem very promising considering the analysis times reported in the article. Nonetheless, the description of the worst-case path through the program via ILP is an elegant method and can be efficient if the size of the ILP is kept small as is the case in our tool.

Lundqvist and Stenström presented an integrated approach to obtain WCET bounds through the simulation of the pipeline in [LS98, LS99, Lun02]. They extend a pipeline simulator to handle unknown values in inputs. With this approach we share conceptual similarities in that we perform a cycle-wise evolution of a pipeline (model). In contrast to our approach, their method is an integrated one, where value analysis for register/memory contents and execution time computation are parts of the same simulation. If the simulation cannot determine a branch condition exactly due to dependencies on unknown (input) values, they have to simulate both branch outcomes. Their method does not guarantee termination of the analysis, but has the advantage of determining loop bounds and/or recursion bounds for free[6]. However, we feel that their analysis is very costly due to the huge amount of data that has to be kept for each branch they follow. In contrast, our method does not keep information like register or memory contents in the pipeline analysis phase. A value analysis can be executed before the cache and pipeline analysis instead. In [LS99] experiments with a PowerPC like architecture are made for small example programs. They use an extended PSIM simulator with simple reservation tables for instructions. In all, it is not clear how well their method scales up to programs of realistic sizes.

Narasimhan and Nilson present a retargetable execution time analyzer for RISC processors in [NN94]. The target architecture is modeled using an extended MARIL language (see [BHE91]). The generated analyzer takes an assembly program and a path represented by a sequence of labels and computes the time needed to execute that path. Due to the use of MARIL the range of targetable processors is significantly limited. Analyzing assembly programs complicates the integration of instruction and data cache analysis. This leads to a large gap between predicted and measured execution time, especially for larger inputs [NN94].

In contrast to Lundqvist and Stenström's integrated approach, Engblom presents a WCET tool with a clear separation of all the analysis modules in [Eng02]. The modules communicate using interface data structures. One main component is a simulator that estimates the execution time for a given sequence of instructions. These timing estimates are composed to form the execution time of the entire program. The quality of the obtained WCET is greatly influenced by the quality of the simulator used. Cache behavior prediction is not incorporated in the tool as the addressed targets do not have caches. This eliminates the problem of cache and pipeline interaction, which becomes more difficult with more complex pipelines, prefetching, and branch prediction. The author comes to the conclusion that "*... out-of-order processors are definitely too complex to model with current techniques.*"

Colin and Puaut describe a framework for tree-based WCET analysis in their paper [CP01]. Instruction cache and pipeline behavior as well as branch prediction are taken into account and are analyzed independently of one another reducing the precision of the obtained WCET estimate.

The analyses are based on two intermediate representations: the syntax tree and the control flow graph built from the assembly output of the compiler. As the program is not yet translated to object code, it is not clear which machine instruction an assembly instruction is mapped to, and as the program is not linked,

[6]If these do not depend in a non-trivial way on unknown input values.

information on instruction addresses is not available. The syntax tree is used to compose the WCET from smaller parts. This is not appropriate as it disregards the execution context leading to imprecise results. In [CP00], they present an analysis to predict the branch processing prediction behavior of a processor with dynamic prediction in the same framework. Modern features like out-of-order execution are not considered. For such architectures, the presented analysis is either not sound or too imprecise (after safety margins have been added to the results).

In [BCP02] and [BBB03] Bernat et al. present a method to obtain probabilistic guarantees for the schedulability and timeliness of a hard real-time system. They argue that WCETs are rarely known and sometimes probabilistic guarantees suffice. In the context our work was done in, i. e. avionics, we have to give hard guarantees for the timeliness of the tasks in the system. So their approach cannot be used in our application area. However, note that a relaxed, much faster version of our analysis that only uses local worst-cases in its computations instead of following all possible scenarios, gives results which are true "most of the time", cf. Chapter 6. I.e. the timing anomalies will rarely occur in practice and local worst-cases will with some probability also be global worst-cases. Thus, our results can be used for the probabilistic method.

Model checking has been widely applied in the area of verification of real-time systems and hardware, [McM93, BCM$^+$92, BBCZ98, McM98, CGP99]. However, its usefulness for determining WCETs seems limited. Although the use of timed automata in principle makes it possible to check a formula which describes the execution of the program under the condition that it finishes before its deadline, this does not classify as a scheduling test under a preemptive scheduling regime (the preemptions by other tasks are not considered). Even then, one must use abstractions to be able to consider all possible states in the timed automata (which depend on input data to the task). To obtain tight WCET bounds, model checking must be repeatedly applied with differing time bounds to check for. How to obtain the formula to check (and prove its correctness) and an adequate timed automaton is another issue in this approach. See [Wil04] for a more in-depth discussion of these questions. In [Met04], Metzner has presented an application of model checking for WCET analysis by analyzing an instruction cache in combination with a simple processor execution model. He argues that model checking can be applied to and indeed improve the precision of WCET prediction by avoiding the abstractions and approximations made by data-flow analyses based on abstract interpretation. It will be interesting to see if this approach scales up to the analysis of data caches and processors with branch prediction, prefetching and out-of-order execution, all features that introduce nondeterminism in the analysis.

In [Fer97] Ferdinand presents a method for static cache and pipeline analysis based on abstract interpretation. Using this methodology, Schneider et al. developed a pipeline behavior prediction [SF99] for the SuperSPARC processor. Both analyses are components of a worst-case execution time prediction. However, their approach uses a simple reservation table scheme in the pipeline analysis which cannot be extended to more complex architectures. Also, they do not consider prefetching and/or branch prediction and assume the sequence of instruction accesses to be known statically. Ferdinand's cache analysis is integrated into our pipeline analysis, adapted to the update semantics of ColdFire's and PPC 750/755 cache .

The problem of scheduling effects on the WCETs of tasks has been studied by Schneider in [Sch02]. There, maximal damage due to task preemptions is computed by using response time analysis with additional damage terms. Our pipeline analysis has been utilized in this work to obtain WCETs for undisturbed executions and then damage costs are added to the results. Also, the effects of the real-time operating system are considered in this work (interrupts, system calls, etc).

Except for the works based on our pipeline analysis, none of the work mentioned above takes into account the combination of branch prediction, instruction prefetching, speculative execution, or effects caused by, e. g., data accesses colliding with wrap-around cache line fills on the external processor bus. Also, modern computers feature separated core and bus clocks, causing new wait effects for external bus accesses, which have not been investigated so far. Most models simplify the memory architecture of the systems they consider; in a real system, the memory space is made up of quite a few regions with different characteristics concerning access timing, access mode[7], and so on. Our tool makes it easy to specify these parameters and the pipeline analysis takes all this into account when examining memory accesses.

1.4.2 Pipeline Modeling

A lot of work exists about hardware modeling, especially of processors and peripherals. Specialized languages (Hardware Description languages, HDL) for specifying hardware have been developed to help in EDA[8]. The most important ones are Verilog [TM91] and VHDL [VHD00]. Both languages allow to automatically synthesize electronic circuits from (restricted) descriptions. The languages are based on the notion of modules, each running a number of *processes*, communicating via *signals*. In the case of VHDL, signals can even carry more complex data, while Verilog basically considers signals as physical wires with a limited set of states (asserted, floating, high impedance, etc). Designs can be tested by *sim-*

[7]E. g., speculative accesses with the PowerPC can be allowed on some regions but disallowed on others.

[8]Electronic Design Automation

ulation of models and models can be formulated at different levels of abstraction (RTL, gate level, etc).

A lot of work has been done on the verification of the synthesis step from models into circuits, and also on obtaining higher level descriptions from lower specifications [AK95], [McM98], [HQR98] or [JPM02]. Little work has been done on the equivalence checking of abstract models with concrete ones, as would be required for checking that the abstractions of our pipeline models are correct. Another area where model checking can be used is to prove that certain timing anomalies are not present in a processor or do at least not occur for a given program. See Section 9.3 for a discussion.

First steps in the area of analyzing HDLs are the papers by Hymans [Hym02, Hym03], which present an abstract interpretation of a VHDL kernel. By adding a monitor and a test bench to an existing model, safety properties can be checked by observing the analysis results for assertions in the monitor, which describe the desired properties (e. g. that two signals are never asserted at once). The analysis is complex on low level VHDL models, but the same kind of analysis should be possible on our pipeline models (which are quite similar to VHDL), and should be much faster due to the higher abstraction level of our pipeline models.

Recently, *SystemC* [Ope02] has been promoted as a modeling framework for embedded systems ranging from processors to complete systems. SystemC is basically a collection of C++ classes, with a predefined simulation engine. As SystemC descriptions are in essence C++ programs, analyzing them is very difficult, as is the analysis of C++ itself. The implicit semantics of the predefined SystemC components (the simulator, etc) has to be abstracted in order to obtain at least a small amount of precision. The correctness of any analysis of a model is difficult to establish due to the complex semantics of C++. Equivalence proofs between models at different levels of abstraction are tedious for the same reason.

Apart from retargetable timing analysis, pipeline descriptions have been used in code generation and simulation. Early work [BHE91] considered only registers and pipeline stages by the use of reservation tables. Improvements have led to mixed-level languages such as Expression [HGG+99]. From a structural specification of hardware resources, pipeline mechanism, and data transfer paths, reservation tables are generated automatically [GHDN99], but this approach is not able to model dynamic behavior like out-of-order execution. Validation of models done in Expression is described in [MTH+02]. Correctness has to be expressed as properties and it is checked if the flow in the model violates this property or not. Due to the form of the description (LISP like), this mechanism is not suited to model complex processors in a maintainable fashion.

Dynamic hardware features are handled in hardware description languages that are used for retargetable simulators. The language LISA [ZPS+96, PZHM98] uses a refined form of reservation tables called L-charts that can model dynamic scheduling to some extent. The language RADL [Sis98] offers a flexible signal mechanism that seems more promising in the context of modern hardware features. Signals are emitted if a boolean expression on machine state components holds; they are used to influence scheduling behavior. In contrast to our work which also deals with signals, RADL describes a model very close to the silicon level, where, e. g., latches between pipeline stages are defined implicitly and may be accessed bitwise.

Chapter 2

Modern Hardware

The need for computing power is ever increasing, resulting in an increasing demand for faster computer systems. The number of transistors that can be put on a chip is on average doubling every 18 months, according to *Moore's Law*, [Moo65][1].

The growing availability of gates per chip has led to the integration of more complex functions into CPUs and higher capacity of main memory. Furthermore, a tighter packing of gates allows to increase the clock speed of the chips, because signal delay times are shortened. The increasing performance of CPUs and main memory is depicted in Figure 2.1: following [HP96], an increase of 7% for memories and of 35% (until 1987) and 55% (from 1987 on) for CPUs is assumed.

The enhanced performance capabilities of CPUs have led to the development of sophisticated features like *pipelining* or branch prediction, in order to be able to actually utilize them for machine code programs. The increasing gap between the performance of CPUs and that of main memories is the main bottleneck for system performance and will continue to be so. CPUs spend most of their time waiting for slow main memory. The cause of the problem is that one can either produce cheap but slow memory (DRAMs) of high capacity or an expensive but fast one of lower capacity (SRAM, register cells). The solution applied today is the introduction of a *memory hierarchy*. Between CPU (core) and main memory a small but fast *cache* memory is placed. Since programs normally exhibit a high degree of temporal and spatial locality, caching accesses for reuse in the fast memory improves the overall performance considerably. Trends nowadays go to two or even three levels of cache memories between the CPU core and main memory.

Where caches are used to reduce access latencies, pipelining is used to make use of the massive parallelism enabled by overlapping the execution of program

[1]The original paper by Moore estimated a doubling of transistors *per chip* every 12 months. The doubling period for *transistors per square centimeter* has flattened to 18 months in the 1970s, cf. [Tuo02, Sto03].

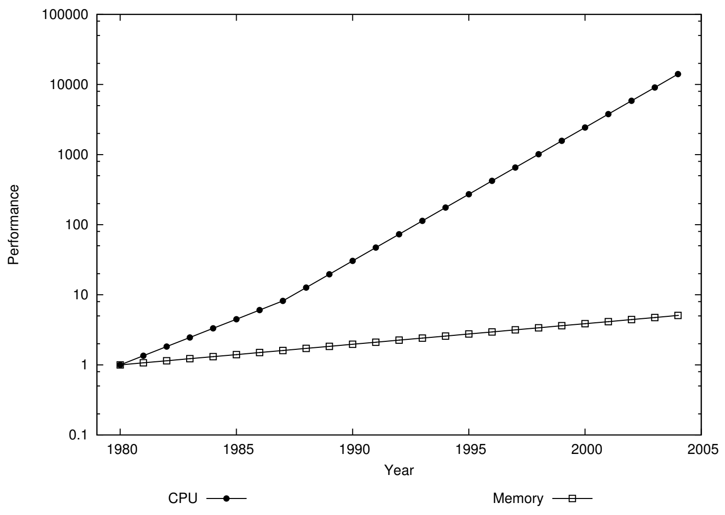


Figure 2.1: Performance gap between CPUs and main memory

instructions. The execution of an instruction can be split into a number of phases like fetching, decoding, executing and writing back, each consuming one clock cycle. The different stages of consecutive instructions can be executed in parallel, bringing the maximal performance to one instruction per processor cycle.

Both caches and pipelines are good in increasing the *average* performance of the system. However, there may be pathological cases of programs that exhibit a very bad *worst-case* performance. Moreover pipelines and caches are very *history-sensitive*: their behavior depends on the instructions executed before. This makes it difficult to predict the behavior of the actual instruction. E.g. whether an instruction cache access is a hit or a miss depends on the instructions fetched before the current one. Whether a pipeline has to stall due to hazards depends on the predecessor instructions still in the pipeline, etc. Another issue is that pipelines and caches may interfere in non-obvious ways such that an event locally classified as a best-case scenario, e. g. a cache hit, may globally lead to the worst-case, i. e. a longer execution time of the whole program. Such contrived effects make it very difficult to argue about the worst-case execution time of programs running on systems with caches and/or pipelines.

In the next section, the architecture and behavior of caches are summarized, followed by an overview of processor pipeline architectures in Section 2.2 and the concepts of other system components in Section 2.3. Afterwards we present the important notion of *timing anomalies* which are the main cause of the complexity of WCET tools for modern processors. Two processors for which we have instantiated the analysis described in this thesis are presented in more detail to give an impression of the sort of hardware features that have been modeled with our approach.

2.1 Caches

A *cache* is a small fast memory that is between the main processor core and slower memory. There can be several levels of caches nested, the one closest to the processor is the L1 cache, the next one the L2 cache and so on. Data that is not found in a cache at level n is searched for in level $n+1$ or finally the main memory. The architecture of a cache is depicted in Figure 2.2.

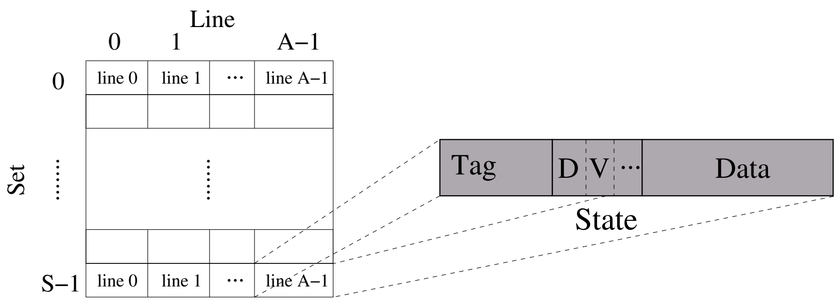


Figure 2.2: Cache architecture

The cache consists of S *sets* with A lines each. A is called the *associativity* of the cache. If $A = 1$ then the cache is called *direct mapped*; if $S = 1$, then it is called *fully associative*, otherwise it is called *A-way set associative*. Each set s_i thus is a fully associative sub-cache in itself. Each line l_j contains a *memory block* of data. The size of this block is called the *line size*, *L*. Apart from the memory block, a *tag* is stored in the line that gives the high-order memory address bits for the memory block stored here. Finally, there are *status bits* in the line that indicate, whether the entry contains valid data ($V = 1$), and for data caches, whether the data in the block has been changed in the cache but not written to main memory, i. e. whether the line is *dirty* ($D = 1$). There can be additional status bits for coherency state or process tags. The parameters characterizing a cache are its *capacity* C, its line size L, the number of sets S and the associativity A, $C = S \cdot A \cdot L$. The line size and number of sets are always powers of 2, $L = 2^l$, and $S = 2^s$. Normally, the associativity and capacity are also powers of two (one exception is the instruction cache of the SuperSPARC, which has $A = 5$ and $C = 20480$, cf. [Mic92]). Table 2.1 shows instruction cache configurations for some processors.

Another parameter is the *replacement policy* of the cache, which determines the lines that should be replaced from the cache when it is full and further data is accessed.

Processor	C	A	L	Replacement
SuperSPARC II	20kB	5	64	LRU
UltraSPARC III	32kB	4	32	LRU(?)
PPC 755	32kB	8	32	Pseudo LRU
G5 (PPC 970)	64kB	1	32	-
MCF 5307	8kB	4	16	Pseudo Round Robin
Pentium III	16kB	4	32	LRU
Intel IXC1000 (XScale Core)	32kB	32	32	Round Robin
SH5-101	32kB	4	32	LRU
Alpha 21264	64kB	2	64	Branch Predict
MIPS 24k	64kB	4	32	LRU

Table 2.1: Some instruction cache configurations

The places where data blocks are stored in the cache are determined by the addresses of the memory blocks. When accessing data at address a, the lower l bits are used to select the data in the 2^l bytes of the line data. The next lower s bits of the address form the index i of the set where the block may reside, thus $i = (a/L)\&(S-1)$.

For data read accesses or instruction fetches, the A elements in the selected set are searched in parallel by comparing the top $w-(l+s)$ bits of the address, where w is the total number of address bits, including the tags of the lines in the set (ignoring lines, whose V bit is not set). If a match occurs, then we have a *hit* in the cache and the data in the line (indexed by the lower l bits of the address) is returned to the processor. If the required memory block is not in the cache, then it must be loaded from main memory. The replacement strategy decides, where the newly loaded memory block should be placed in this set. Normally, invalid lines ($V = 0$) are filled with new data first. If there are no invalid lines, a line is selected to be replaced from the cache and the new data takes its place. Which line is replaced is determined by the *replacement strategy*. When new data is placed into the cache, the tag of the line is updated with the tag bits of the access address, the *V* bit is set and *D* is cleared (for data caches). Then the processor starts to fetch the bytes of the data for this line from the memory hierarchy (next cache level or main memory). As the line size is normally bigger than one data word, it takes several external accesses to fill the complete line. If the data word that was referenced in the line is fetched first, the fetch is called *critical word first* (and wraps around at the end of the line). Otherwise the first word of the line (offset = 0) is fetched first. If the cache can serve (hit) accesses while the remainder of a line is being loaded it is called a *hit under miss* cache.

For data writes, there are several possibilities for the behavior of the cache. If the write is always issued to main memory (changing the corresponding data block in the cache, if it happens to be there), the cache is called *write-through*. If the data is only changed in the cache, it is a *write-back* cache. In the latter case, the data must eventually be flushed to memory to obtain a consistent state. This is done either by explicitly flushing the cache through machine instructions or when the modified line is replaced from the cache due to another cache miss. To record if a line has been written but not flushed to memory, the cache line has the *dirty* bit *D* set to 1. If a written-to block is not in the cache, the block may either be loaded from memory first, placed in the cache and then modified (*write allocate*) or the data may just be written to main memory, without loading the line. Normally, write-back and write-allocate are used together.

Some architectures allow to *lock* several ways of the cache. Then, no replacement occurs in these portions, but data may be loaded in the first access to empty (invalid) lines of the way. This feature is often utilized to preallocate critical code and data for fast and deterministic access.

All architectures allow to designate only portions of their address space as cacheable, so that only accesses to these areas go through the cache. The other memory areas bypass the cache completely and always result in external bus transactions.

A cache that holds both data and instructions is called a *unified* cache. Here, data reads and writes compete with instruction fetches for entries in the cache. Most systems nowadays internally have separate paths for data and instructions (Harvard architecture) and the paths are connected to separate data and instruction caches.

2.1.1 LRU Replacement

One replacement strategy that is often implemented is the *least recently used* strategy. The line in a set that has been unreferenced for the longest time is the one being replaced first. Here, the sets of the cache are independent, i. e. the references of lines are counted separately for each set. An age of the A lines in a set is introduced such that the line referenced last is the youngest line and the one unreferenced for the longest time is the oldest. Each access to the set, cache hit and miss, updates the ages of all lines. If the access is a cache hit, the line accessed becomes the youngest line (age 0), and all lines that were younger than this line before this access age by 1. For a cache miss, the oldest line is replaced from the cache, all other lines age by 1 and the newly loaded line becomes the youngest line. Figure 2.3 shows a set with 4 ways and the aging of its lines for a cache hit and a cache miss.

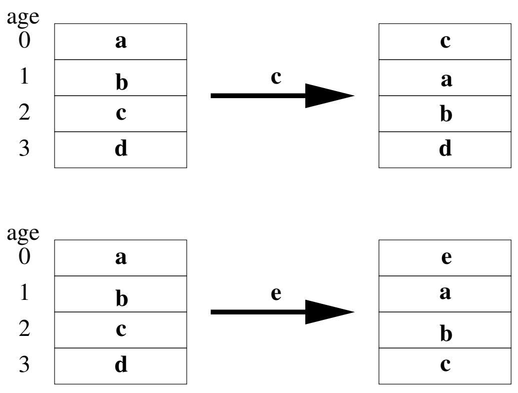


Figure 2.3: Aging of cache lines under accesses for LRU

To describe this strategy precisely, we can define an update function for a cache. The set of memory block addresses M is the set of integers obtained from addresses by leaving out the lower l bits, which only select data in a line. We describe the cache contents by giving for each memory block address its age in the set it is mapped to. For blocks that are not in the cache, we introduce the "age" $\top$. So the set of ages is $G = \{0, \dots, A-1, \top\}$. We introduce an ordering $\leq$ on them by $0 \leq 1 \leq \dots \leq A-1 \leq \top$. An addition $\oplus$ on G is defined by

$$a \oplus b = \begin{cases} \top & \text{, if } a = \top \vee a = \top \vee a+b \geq A \\ a+b & \text{, otherwise} \end{cases}$$

The set a memory block is placed in is given by the function $\mathrm{set} : M \to \{0, \dots, S-1\}$, defined by $\mathrm{set}(m) = m \bmod S$. We write $\mathrm{set}_i(M)$ for the set of memory blocks mapped to set i.

A *cache* is now a mapping $C : M \to G$ which maps memory block addresses to their age. Since sets in the cache are independent, we introduce a mapping of sets to ages, i.e. a set mapping T_i is a function $T_i : \text{set}_i(M) \to G$. Then a cache is defined in terms of S set mappings: $C(m) = T_{\text{set}(m)}(m)$.

A set mapping T_i must satisfy the constraint $T_i(m) \neq \top \Rightarrow \nexists m' : m' \neq m \wedge T_i(m) = T_i(m')$, i. e. there cannot be different memory blocks with the same age *in* the set.

The *update* of a cache when accessing a memory block m is then described by a function $\mathcal{U}^c : (M \to G) \times M \to (M \to G)$ with

$$\mathcal{U}^c(C,m)(m') = \begin{cases} C(m') & \text{, if } \text{set}(m') \neq \text{set}(m) \\ \mathcal{U}^s(T_{\text{set}(m)}, m)(m') & \text{, otherwise} \end{cases}$$

The update of a set mapping, $\mathcal{U}^s(T_i, m)$, is given by

$$\mathcal{U}^s(T_i,m)(m') = \begin{cases} 0 & \text{, if } m = m' \\ T_i(m') & \text{, if } m \neq m' \wedge T_i(m) \neq \top \wedge T_i(m) \leq T_i(m') \\ T_i(m') \oplus 1 & \text{, otherwise} \end{cases}$$

2.2 Pipelines

The process of executing one instruction can be divided into a number of sequential sub-operations. A simple example is the pipeline of the DLX machine from [HP96]:

- **IF**: Instruction fetching from memory
- **ID**: Instruction decoding and fetching of register operands
- **EX**: Execution of the operation denoted by the instruction
- **MEM**: Memory access (read or write)
- **WB**: Write-back of the results into registers

Each of these sub-operation is called a *pipeline stage*. Instead of waiting until an instruction has left the last stage of the pipeline (we say it has *retired* in this case) before letting the next instruction entering the first stage, as depicted in Figure 2.4, one can *overlap* the execution of different stages of subsequent instructions. If there are no dependencies between stages, a perfect pipelining as in Figure 2.5 is achieved. Here, the execution of an instruction takes one cycle, as soon as the pipeline has been *filled* completely with instructions. We say that the *CPI* (clock cycles per instruction) is equal to 1, in this case.

Stage / Clock Cycle	IF	ID	EX	MEM	WB	Finished
1	I_1					
2		I_1				
3			I_1			
4				I_1		
5					I_1	
6						I_1
7	I_2					

Figure 2.4: Sequential execution

2.2.1 Pipeline Hazards

Pipelining exploits the parallelism that is inherent in a program. In practice a CPI of 1 is never achievable as there are dependencies between instructions and other *hazards*. When a hazard occurs, parts of the pipeline have to be *stalled*, i. e. some stages do not perform any work but wait until the stall condition has vanished. Pipelines differ in which stages are stalled by a hazard: while some stall all stages before the one the hazards occurs in, others only stall the stages that want to

Stage / Clock Cycle	IF	ID	EX	MEM	WB	Finished
1	I_1					
2	I_2	I_1				
3	I_3	I_2	I_1			
4	I_4	I_3	I_2	I_1		
5	I_5	I_4	I_3	I_2	I_1	
6	I_6	I_5	I_4	I_3	I_2	I_1
7	I_7	I_6	I_5	I_4	I_3	I_2

Figure 2.5: Fully pipelined execution

deliver results to stages stalled by the hazards. This way, e. g., instruction fetching to a prefetch queue could continue if a data hazard stalls instruction dispatch due to data dependencies. There are three categories of hazards:

Structural Hazards

These hazards result from a shortage of *functional units* in the processor. They occur when two instructions in the pipeline try to use the same unit in the chip. Often the memory bus is the functional unit that causes these hazards: the IF and MEM stages both want to transfer data over the bus. In this case, IF must be stalled until MEM has finished the transfer. This inserts a *bubble* after the IF stage, i. e. the ID stage is empty in the next cycle and performs no work. Another example are the integer units of the PowerPC 755: if more than 4 integer instructions have been issued, the issue of the next instruction must be stalled since there is no free integer unit (or reservation station) available for it.

Data Hazards

These hazards are the most common ones and originate from data dependencies between the operands of subsequent instructions. E.g. in Figure 2.6 the second instruction needs the result of the first one as an input operand in register **r8**. The second instruction cannot begin its execution until the first one has written back the result into the register file.

```
0x10000:  add r8,r9,r9
0x10004:  sub r10, r8, r11
0x10008:  add r11,r7,r7
0x1000C:  add r6,r11,r11
0x10010:  add r11,r8,r8
```

Figure 2.6: Data dependencies between instructions

There are three different kinds of data hazards between instructions:

- *Read after write* (RAW) hazards occur if a subsequent instruction reads a register that is written by a previous instruction, as for the first two instructions in Figure 2.6.

- *Write after read* (WAR) hazards occur if an instruction writes a register that is read by a previous instruction. Here it must be guaranteed that the write occurs only after the first instruction has read the register. This is guaranteed for the simple pipeline of the DLX, due to its in-order structure. For out-of-order execution, like the PowerPC 755, special logic must be implemented to resolve this hazard. In Figure 2.6 the instructions at 0x10004 and 0x10008 cause a WAR hazard.
- *Write after write* (WAW) hazards occur if a subsequent instruction writes the same register as a previous one. It must be guaranteed that only the last write is performed to the register file. Instructions 0x10008 and 0x10010 cause a WAW hazard in the example. Again, for in-order pipelines with only a single write-back stage, this hazard cannot occur. On the PowerPC 755, the reorder buffer and its in-order retirement regime guarantee that the ordering of writes to the register file is maintained.

Control Hazards

The pipeline only delivers good performance, if it is filled with instructions. If a dynamic branch instruction occurs, either as a conditional branch or an indirect jump, the fetch stage cannot continue to fetch the next instruction after the branch until the branch target is known, i. e. until the instruction that computes the argument to the branch has written back its result. This means that the pipeline will run empty and has to be refilled with the instructions at the target address when the depending instruction has been completed.

2.2.2 Performance Improving Features

Several features have been implemented in pipelines to avoid hazards or limit their impact on (average-case) performance.

Prefetching

A pipeline can be designed to continue instruction fetching during a stall in later stages of the pipeline caused,e. g., by a memory access into a *prefetch queue*. Instructions can then be decoded and *issued* from this queue, eliminating delays due to instruction fetching. The prefetch queue can either hold decoded instructions (as for the MCF 5307, with 8 entries in the prefetch queue) or raw instruction words (as for the PowerPC 755 with 6 entries in the instruction queue). Care must be taken to flush the contents of prefetch queues when data in instruction memory has changed. Prefetching introduces another complication for WCET analysis, as the amount of prefetching depends on the overall state of the pipeline: if the pipeline stalls due to data dependencies, the prefetch queue fills up with instructions. Since instructions are normally fetched via the instruction cache, this (in combination with branch prediction) alters contents and ages of lines in the cache. Which, in turn, influences the replacement behavior of the cache and thus the global timing.

Branch Prediction

To reduce control hazards the pipeline should be filled even after a branch instruction has been fetched. This can be done by combining prefetching with decoding of branches early in the pipeline and redirecting the fetching of instructions to the known target of a static (non-dynamic) branch (*branch folding*). If the branch depends on other results, branch folding can also be applied when the result is already known during fetching/decoding of the branch instruction. If the result is not yet known, one can *predict* the target of the branch in the case of conditional branches, which only have two possible target addresses (the instruction following the branch and the instruction at the fixed taken-target address of the branch). Fetching of instructions is redirected to the predicted target address of the branch, based on the assumption that after the result of the branch condition is known, it will resolve the prediction as correct and the instructions are already available. If the prediction went wrong (*misprediction*), the instructions fetched are discarded and the fetching is redirected to the correct target address of the branch (and the pipeline stalls until the instructions are available).

The prediction can be done in a variety of ways:

- *Static* branch prediction encodes the most probable target of the branch in the branch instruction. E.g. for the MCF 5307, backward branches are predicted as taken, as they correspond to the loop back edges and are usually taken N out of the $N+1$ times the condition of a loop with N iterations is evaluated. For other architectures, a special bit in the branch instruction can indicate, whether the branch should be predicted taken or not to allow for compiler optimizations of branch instructions (one example is the PowerPC architecture). Static branch prediction has the advantage that it is easy to implement in hardware and also rather easy to account for in a WCET analysis.

- *Dynamic* branch prediction uses a cache to record the last outcomes of branches and based on this history it predicts the next branch direction. Most architectures with dynamic branch prediction feature a *branch history table* (BHT) that is organized as a little cache where each entry records the address of a branch instruction and a *branch counter* of k bits. If the value of the counter is $\geq 2^{k-1}$ the branch is predicted as taken, otherwise it is predicted as not-taken. In the case of a misprediction, the counter corresponding to the BHT entry of the branch is decremented by one, saturated at 0, otherwise it is incremented by one, saturated at $2^k - 1$. If $k > 1$, then this prediction scheme reacts less sensitive to one misprediction in a series of correct predictions. Usual values for k range from 0 to 4. The PowerPC 755 has a 512 entry BHT with $k = 2$ bits. Another form used, often in combination with the BHT, is the *branch target buffer* (BTB). This is also a cache of recent branches but instead of just containing a prediction for the outcome of the branch, it stores some instructions at the predicted target address of the branch. If a branch hits in the BTB, the instructions at the predicted target address are immediately available to be inserted into the instruction fetch stream. After a branch is first encountered, the instructions fetched after the prediction are put into the BTB. The PowerPC 755 has a 64 entry BTB (called BTIC here). The advantage of the BTB is that the instructions are available earlier than they can arrive from the instruction cache (usually one clock cycle).

Static branch prediction proves to be very efficient in practice with less than 30% mispredictions according to [HP96]. While dynamic branch prediction is more accurate on average, the fetch behavior of programs depends on the execution history (the states of the BTB and BHT) and is thus much harder to analyze than with static branch prediction. To obtain a precise WCET, the branch prediction must be modeled precisely, which makes the resulting analysis more complex.

Delay Slots

A different method to avoid pipeline stalls due to dynamic branches is to use one or more *delay slots*. Delay slots are the instructions directly following a branch. These instructions are always executed, even if the branch is taken and the program execution continues at a different place in the program after the delays slot instructions have been executed. This way, the pipeline need not be stalled if the condition of the branch is not yet known. Ideally, the condition of the branch is known while the delay slot instructions have been executed and no further delay occurs. The SPARC architecture, e. g., has one delay slot. Delay slots make it a little bit more difficult to reconstruct control-flow from a binary executable.

Forwarding/Shortcuts

RAW hazards can be partially eliminated in a pipeline by *forwarding* the result of an operation to any other stage that holds an instruction depending on this result.

This way, the depending pipeline stage need not be stalled until the previous instruction reaches the write-back stage. *Shortcuts* are special-case evaluations for instruction execution, e. g. a multiplication can be shortened if one argument is 0 (or has the upper $8 * n$ bits set to zero), which means that execution times for instructions depend on operand values. Forwarding can be viewed as an instruction independent shortcut mechanism. Forwarding and shortcuts together with *timing anomalies* (see Section 2.4), can make a WCET computation expensive, as every possible execution duration has to be considered.

Superscalarity

The simple DLX pipeline described above can only decode and start to execute one instruction per clock cycle, thus the optimal CPI is equal to 1. *Superscalar* (or *multiple-issue*) machines can start the execution of more than one instruction per clock cycle. If a processor can issue two instructions at once, the ideal CPI is 0.5 (the PPC 970 can issue up to 4 instructions plus one branch per cycle, giving a ideal CPI of 0.2). How many instructions are issued is decided dynamically, according to dependency rules among the instructions in the prefetch queue. Naturally, the dynamic nature of multiple-issue makes is harder to precisely analyse it for WCET, as the modeling complexity increases.

Out-of-order Execution

In order to overcome stalls due to structural and data hazards, a processor may execute instructions *out-of-order*. That is, the execution of subsequent instructions may be begun and finished before a previous instruction has finished its execution. This greatly increases the utilization of the functional units in the processor, as later instructions with no data dependencies can already be issued and dispatched to the functional units while their predecessors are still waiting for input data or continuing their execution. E. g. an integer division usually takes more than one clock cycle and blocks subsequent instructions from executing in a strict in-order pipeline.

In a processor with multiple-issue, the distribution of instructions to functional units can be out-of-order too: which instruction gets to a functional unit and which one has to wait in a reservation station can be decided based on availability of operands. On the other hand, like in the PowerPC 755, issuing can be in-order but only the execution out-of-order.

Out-of-order execution makes the design of pipelines much more difficult, because correct reading and writing of results in program order must be maintained. Also, the problem of *precise exceptions* arises. If one instruction causes an exception, e. g. a page fault for a memory access, then the operating system must know exactly where to restart program execution after the page fault has been handled. In an out-of-order pipeline it is not immediately clear how to restart program execution if an instruction causes an exception while predecessor instructions are still unfinished.

Two approaches of maintaining a correct machine state for out-of-order pipelines are the *scoreboard*, [Tho64], and the *Tomasulo Algorithm*, [Tom67]. The latter one uses reservation stations at functional units that take instructions waiting for their input dependencies to resolve and a *reorder* buffer that keeps track of the sequential dependencies of register updates.

For WCET analysis, out-of-order execution poses big challenges because of its highly dynamic nature with great influence on caches and memory access behavior.

Superscalarity and out-of-order execution also make it harder to exactly define the notion of "execution of an instruction", as a dynamically varying number of instructions begin and end execution per clock cycle.

Speculative Execution

The performance gained by out-of-order execution is reduced if a branch is encountered, whose condition is not yet known, as the instructions at the (predicted) target address of the branch cannot be executed (issued). We call instructions at the predicted target address of a branch *speculative instructions* until the branch outcome is resolved.

The Tomasulo approach can be extended to allow speculative instructions to begin their execution, i. e. being issued to functional units. If a prediction turns out to be wrong, the speculative instructions are simply flushed from the units and their entries in the reorder buffer are freed. Speculative instructions are not allowed to write-back their results into the register file until the predicted branch that caused them to be speculative has been resolved and the prediction affirmed.

Naturally, speculative execution makes the analysis of the pipeline behavior even more complicated and necessitates a more complex modeling of the structure of the pipeline in the analysis.

2.2.3 Other Features

Besides the features implemented in the CPU to enhance performance, there are some more features implemented to support the operating system or to guarantee a deterministic system state from the programmer's point of view. In the following, we present some of these features that may have an impact on WCET prediction.

Memory Protection

In order to support an OS that supplies multi-process, multi-user capabilities, protected memory areas must be provided by the CPU. Here, every process in the OS has the impression that it runs alone on the CPU in its own address space. This is implemented by using *logical* addresses in the processes, which are translated to *physical* addresses by the *virtual memory* unit of the CPU. When switching processes, the OS also switches this mapping, which is sometimes called *VM mapping*.

The VM mapping is stored as an array of descriptors (PTE, Page Table Entries) in memory, each descriptor mapping a *page* of fixed size (mostly 4kB) from logical to physical address. If an access to a page occurs, the CPU has to access the corresponding PTE in memory to obtain the physical address for it, before it can perform the access (possibly through the cache). To increase performance, the CPU keeps a small cache of recently used mappings, so that further accesses to a page whose mapping is in the cache can be performed without loading its associated PTE from memory. The capacity and associativity of this cache, which is called *Translation Lookaside Buffer* (TLB), varies widely. Usually, LRU is used as the replacement strategy for entries in the TLB. Apart from the mapping of logical to physical address of a page, the PTEs also store *access attributes* for the page, i. e. if the page can contain instructions, can be written to, is cacheable, etc. Accesses with attributes not matching the page attributes lead to program exceptions, as would accesses for which no mapping has been established. For pages containing data, the PTEs also record if this page has been written to by the processor, so that the OS can write back dirty buffer cache pages to files. If the processor writes to a page, the associated PTE is fetched (if not present in the TLB) from memory and a modified copy, which has the dirty bit set, is then written back to memory before the write access is performed. Thus, a write access can lead to two or more write accesses to memory if virtual memory is used. The write-back of the PTE need only be performed for the first write access to the page.

For systems supporting I/O operations via dedicated pins (like the Intel architecture), a similar feature exists for controlling access to the I/O address space.

Some systems, like the MCF 5307, do not provide full virtual memory support but can also associate certain areas of addresses with certain attributes. These attributes then determine if a memory area is cacheable, writeable, executable, etc. This feature is also present as an additional VM mapping mechanism in systems supporting PTEs and TLBs to map large areas whose attributes never change (like OS kernel address space), e. g. the IBAT and DBAT mapping registers of the PowerPC family, cf. Section 2.6.

Since the memory access timing with enabled virtual memory depends on the contents of the TLB and the status of the page (first write to a page), its behavior is important for WCET determination. As addresses for data accesses are sometimes not precisely known in a WCET analysis, the behavior of VM is hard to predict for them. As VM is rarely used in real-time systems, most WCET prediction techniques assume that it is turned off, leading to more predictable access behavior.

System Configuration

On systems embedding peripherals in the CPU, special control registers are usually available, which determine at which addresses the peripherals can be accessed. E.g. the MCF5307 has an embedded SRAM memory area of 4kB which can be accessed via the fast processor core bus. The memory address of this area must be programmed into a system register. Other registers, possibly memory mapped, control the peripherals (UARTs, DMA engines, etc).

The WCET tool for the MCF5307 must know about the address of the SRAM area, as accesses to it are faster than accesses to external memory. Also, accesses to peripheral registers are faster than external memory accesses, so the register areas of the peripherals must be known.

Although the mappings can be changed dynamically by writing new values to the associated system registers, this is rarely necessary in real-time systems which have a fixed configuration. Thus, such dynamic changes need not be modeled by WCET tools.

Synchronization

In a highly parallel pipeline, changes to system registers or VM mappings may not be able to affect instructions already (partially) being executed. To ensure that after a given point of program execution, all subsequent instructions can rely on the new system state imposed by such changes, *synchronization* points are required. Some architectures, like the PowerPC, define synchronization properties of certain instructions. Upon fetching such an instruction, the CPU automatically prevents the execution of subsequent instructions or even refetches an instruction stream to ensure that all effects by the instruction to the system state are committed or that no two instructions alter the system state out of their program order.

Naturally, such behavior alters the timing of program execution quite significantly compared to a fully pipelined program run. In combination with instruction prefetching and speculation this can also have profound impacts on the ages or contents of cache blocks. A WCET tool must therefore take the synchronization behavior of instructions into account.

2.3 System Components

Not only the caches and pipeline of the processor have an influence on the timing behavior. Also the other (computer) components of the real-time system must be considered. This section will present some components and their influence on system performance and thus WCET analysis. In our WCET tool, we have modeled and integrated several of these components to obtain a correct and precise analysis of the whole system.

2.3.1 Memory

The component with the greatest impact on execution time is the main memory. Access to main memory is performed via a memory controller that translates bus transactions of the processor to the access protocol of memory chips. In some cases, this controller is embedded directly into the CPU (MCF 5307 for example) or has to be provided by external circuitry. The RAM controller is commonly integrated into the main *system controller* (called *northbridge* in the PC world) that also contains the interfaces to the peripheral busses.

SRAM

Static RAM (SRAM) implements the single RAM bit cells by transistors and thus is quite fast. Little extra logic is needed to interface SRAM to the common address and data buses of CPUs. On the other hand, SRAM is quite expensive and high capacity chips are not easily available. The access times to SRAM are fixed, which provides for good predictability and eases WCET prediction. SRAM systems are only used for small memories and/or if the price of the system is irrelevant.

DRAM, SDRAM

Dynamic RAM (DRAM) has only one capacitor for each RAM bit cell, augmented with a row of transistors, the *sense amps*, that amplify and drive the signals of a row of RAM cells on the data bus. This design allows for extremely tight packing of RAM cells and thus cheap and huge memories. The drawback is the slow speed of the DRAM technology (around 10-16 times slower than SRAM). Internally, the memory chips are organized as arrays of bit cells, with 4096 or 8192 bits in each row and a varying number of columns (up to 4096). Each of these arrays contributes 1 bit to the output data path of the chip, which has between 4 and 64 data bits. So, for a 16 data bit chip, 16 RAM cell arrays are accessed in parallel in the chip. In addition, each chip has a number of parallel *banks*, which duplicate the array structure. Each data array has a row of sense amps, 4096 bits wide, which takes the contents of one row of the array upon reading/writing. The access protocol to these chips differs slightly for some implementations, e. g. EDO (extended data out), FPM (fast page mode) or SDRAM (*synchronous DRAM*).

Access to an SDRAM chip is synchronous to the rising edge of the memory bus clock and can be divided into 3 phases, cf. [Mic03, Int99]:

1. One row of an array in the chip must be copied into the sense amps. This is done by selecting the row with 12 *address inputs* and 2 *bank select inputs* and asserting the $\overline{\text{RAS}}$ (row address strobe) signal. After a certain amount of time (t_{RCD}), the sense amps are filled with the contents of the row from the memory array (which loses its contents). No access must be performed during this waiting phase.

2. After t_{RCD} has passed, a *column* in the array is selected, which means that the corresponding bit in the row of sense amps is selected. This is done by asserting the $\overline{\text{CAS}}$ signal. After a certain amount of time (t_{CL}), the data has been transferred from the sense amps into the output buffers and is available at the data outputs. From now on, new data is available each clock cycle, i. e. the SDRAMs work in burst mode. For write accesses, the data can be transferred each cycle over the bus after $\overline{\text{CAS}}$ has been asserted.

3. After the burst is finished and before the next access to a different row can begin, the contents of the sense amps must be copied back into the capacitors of the memory array (*precharging*). This procedure takes a certain amount of time (t_{RP}).

An SDRAM controller can keep rows (which are also called *pages* in the corresponding literature) in an SDRAM chip *open*, i. e. skip the precharge step until it opens a new row. An access to an open page can skip the first step of the access procedure above and thus is faster (by t_{RCD}). Precharging times can be hidden if the access after the current one (which necessitates the precharge) goes to a different memory chip. SDRAM controllers implement different strategies for keeping SDRAM pages open and mapping memory access addresses to memory banks containing SDRAM and mapping the addresses to the column and row indices of the memory arrays.

In addition, the contents of the DRAM chips have to be refreshed periodically, because the capacitors in the arrays lose their current and thus their bit content. Refreshing is done by copying the rows into the sense amps and writing them back. For this, the SDRAM controller has to issue precharge commandos to the chips periodically (and close any open pages before). During refresh periods, the chips cannot be accessed. Different chips allow different refresh regimes: refreshes distributed evenly for all rows along the refresh interval, or a burst of 4096 refreshes in sequence.

All this makes the access times to SDRAM quite unpredictable. Even without considering refreshes, the open pages in the SDRAMs must be modeled precisely in order to predict the timing behavior of the memory. Although SDRAM is less predictable than SRAM, it is used more often in current real-time system designs due to its price and capacity advantages. For custom-built SDRAM controllers, the behavior of the controller w.r.t. open page and interleaving policies and refresh strategies is mostly known. If a widely available standard controller is used, this is most often not the case, posing additional problems for system predictability and WCET analysis.

2.3.2 Peripherals

A real-time system has a number of sensors to receive inputs and actuators in order to pass the computed actions to the environment. These are realized as some peripheral chips which are accessed by the CPU either via special I/O instructions or normal load/store instructions for memory-mapped peripherals.

A system controller manages the translation from the CPU memory bus to the address mechanisms used by the peripherals. Access times (as viewed from the CPU) for peripherals are mostly constant and thus nicely predictable for WCET analysis. One problem for WCET analysis is that sometimes the analysis cannot exactly determine the address of a data access (due to the static nature of the analysis). In this case, the analysis must also consider that the access will go to a memory-mapped peripheral, which results in increased complexity of the analysis and probably worsens analysis results, as peripheral access times are large (even compared to the slow main memory). A WCET analyzer must provide a means to specify which memory areas are mapped to peripheral access registers.

2.3.3 DMA, Multiprocessors

Direct memory access (DMA) by other peripherals sharing the same main memory as the CPU (or by other CPUs in a multiprocessor system) causes conflicts for concurrent memory accesses of the CPU. As only one device can access the memory, the other(s) have to wait. As DMA activity runs in parallel with the program execution and is not synchronized with it, this means that memory access time can vary according to access blockage. If the main memory is built from SDRAM, the accesses of the peripherals also cause changes in what pages the SDRAM controller has opened, introducing further unpredictability for CPU memory access times.

Cache coherency is another problem: some processors *snoop* the memory bus for accesses of other components to memory and implement a coherency protocol that enforces that two CPUs with caches have a coherent picture of the state of their caches w.r.t. main memory. E.g. if a dirty line exists in one cache and a second CPU writes data to memory at that address, than this line must not be written back by the first CPU to memory, it must be invalidated from the cache. Otherwise, incorrect data would end up in memory. Clearly, snooping influences the cache behavior of processors and reduces the predictability of memory accesses, making a correct and precise WCET analysis difficult.

2.3.4 Busses

Computer system components are connected by a collection of *busses*. Besides the memory bus of the processor, there is the SDRAM bus and busses to peripherals or FLASH memory. Some of the peripheral buses have quite complex architectures. One example is the wide-spread PCI (Peripheral Component Interconnect) bus, [SA95]. Accesses on the PCI bus result in one of three cycles:

1. *Config space cycles* access the configuration registers and information that allow to auto-configure PCI devices.

2. *I/O space cycles* access data in the I/O address space of the peripheral.

3. *Memory space cycles* access data in the memory address space of the peripheral.

On CPUs that don't have I/O space instructions themselves (like the MCF 5307), the second type of cycles is performed by memory access operations which are translated by the system controller to PCI I/O operations. The same is true for the config space accesses. Furthermore, the addresses of the PCI memory and the addresses of the CPU need not be in a 1:1 relation, but are rather mapped by the system controller.

Since the timing of the different accesses for the PCI bus is important for WCET determination, the WCET tool must know about the memory mapping for PCI accesses. Things get even more complicated, if multiple PCI busses are nested, as then transactions are forwarded with yet another access protocol from one bus to the other.

One effect that occurs if multiple busses are present is that in most cases they do not use the same clock speed to time accesses. Nowadays, the CPUs internal clock (core clock) speed is nearly always a multiple of the clock speed of its memory bus. And peripheral busses are also clocked at lower speeds than this memory bus. This means that accesses crossing busses incur *jitter*, i. e. the access has to wait for the next rising clock edge of the target bus before continuing, which results in up to $n-1$ cycles of the faster clock (if the ratio between the clock speeds is n). Sometimes busses are clocked at speeds that are not integer multiples of each other but *half-cycle* multiples, e. g. the core clock running at $4.5\times$ the speed of the memory bus clock. This gives even more jitter possibilities.

2.4 Timing Anomalies

The interaction of several features in a pipeline can be very hard to predict. While all these features try to increase (average) performance, there are cases where they influence each other in such a way that a locally faster execution of an instruction can lead to a *globally longer* execution time of the whole program. More generally, if a change ΔT_1 of the execution time of one instruction leads to a change ΔT of the global execution time we say that a *timing anomaly* (cf. [Lun02]) occurs if either:

- $\Delta T_1 < 0$, i. e. the instruction executes faster, and $\Delta T < \Delta T_1 \vee \Delta T > 0$, i. e. the overall execution is accelerated by more than the acceleration of the instruction or it runs longer than before
- $\Delta T_1 > 0$, i. e. the instruction takes longer to execute, and $\Delta T > \Delta T_1 \vee \Delta T < 0$, i. e. the overall execution is extended by more than the delay of the instruction or the program takes less time to execute than before

The case $\Delta T_1 < 0 \wedge \Delta T > 0$ is a critical case for WCET analysis while $\Delta T_1 > 0 \wedge \Delta T < 0$ is critical for BCET analysis. These cases make it impossible to use local worst-case scenarios for WCET or BCET computation. This necessitates a conservative approximation to the possible damages of all cases or forces the analysis to follow all possible scenarios.

A similar effect is well-known in scheduling theory. E.g., [Gra69] shows that tasks scheduled on multiprocessors can meet their deadlines, if the real execution times are assumed to be the WCETs, but there may be tasks missing their deadlines if they execute for less than the WCET.

Unfortunately, as [LS98, Lun02] have observed, the worst-case penalties imposed by a timing anomaly may not be bounded by a globally fixed constant, but may depend on the program size. In Figure 2.7 an example from [Lun02] is shown that demonstrates that a cache miss may not be the global worst case.

Here, the target architecture is a simplified PowerPC with two integer units. The first integer unit (IU) can execute additions and subtractions in one cycle, while only the second one (MIU) can execute multiplication and division but needs 4 cycles for such an instruction. The load store unit (LSU) executes the load and store instructions. The architecture is supposed to have out-of-order issue and execution. In the example program, instructions 2 and 3 are issued to IU, while 4 and 5 are issued to MIU and instruction 1 to LSU. There exist RAW

```
1  LD    r4,0(r3)
2  ADD   r5,r4,r4
3  ADD   r11,r10,r10
4  MUL   r12,r11,r11
5  MUL   r13,r12,r12
```
Example program

Cycle	LSU	IU/Res	MIU/Res	LSU	IU/Res	MIU/Res
1	1			1		
2	1	/2		1	/2	
3		2/3		1	3/2	
4		3/	/4	1	/2	4/
5			4/5	1	/2	4/5
6			4/5	1	/2	4/5
7			4/5	1	/2	4/5
8			4/5	1	/2	5/
9			5/	1	/2	5/
10			5/	1	/2	5/
11			5/		2/	5/
12			5/			
	Cache hit			Cache miss		

Figure 2.7: Timing anomaly example

dependencies between instructions 1,2, instructions 3,4 and 4,5. If the load in instruction 1 hits in the cache, it takes two cycles to complete, otherwise it takes 10 cycles. In the case of a cache hit, instruction 2 is scheduled to the unit in cycle 3, while instruction 3 goes to the reservation station. Instruction 4 has to wait until instruction 3 finishes, before it can begin execution in cycle 5 due to the RAW dependency. Thus the program finishes after cycle 12 in this case. If the load is a cache miss, then instruction 2 is not scheduled to IU, but instruction 3 is scheduled first in cycle 3, due to the RAW dependency between instructions 1 and 2. In this case, instruction 4 begins execution one cycle earlier in cycle 4, because instruction 3 is finished after cycle 3. Here, instruction 2 overlaps execution with instruction 5 in the last cycle (11) and the whole program finishes one cycle earlier than in the case of the cache hit. This example also shows that discovering timing anomalies can be quite complex.

Some sufficient conditions for the absence of anomalies have been presented in [Eng02] for simple processor designs. But, most modern processors show timing anomalies, even relatively simple ones (like the Motorola ColdFire 5307, which we will present in Section 2.5). Even more disturbing is that it is very difficult to prove the *absence* of timing anomalies. Interactions that cause these anomalies can be quite complicated. In order to show that no anomalies can occur, a precise model of the processor must be made and analyzed. It is a topic of ongoing research to utilize abstract interpretation and model checking techniques for this. Even knowing one timing anomaly scenario for a processor doesn't make it easier to find *all* anomalies for the processor, as they may be caused by different interactions. Furthermore, the interactions may also reach over the processor itself and include the attached peripherals and memories.

We will present examples of timing anomalies for two processors in the two sections describing the MCF 5307 and the PowerPC 755.

2.5 The Motorola ColdFire 5307

The Motorola ColdFire 5307 (or MCF 5307 for short) is a member of the version 3 ColdFire family from Motorola. The ColdFire family of processors is a successor to the well known 68k processor series from Motorola. The MCF 5307 understands most of the instructions of the 68030 processor but restricts the instruction length to be either two, four, or six bytes (one to three words), compared to the up to 24 bytes of an 68030 instruction. In addition, a few instructions are not supported. But in general, most of the existing code for M68k designs can be reused.

As a member of the V3 ColdFire family, the MCF 5307 shares the general structure of that family, which is shown in Figure 2.8.

Internally the MCF 5307 has a hierarchical design of busses to access data. The processor core, which features the pipeline and a MAC unit, is connected via a pipelined bus, the so called K-Bus, to the cache and an on-chip SRAM module of 4kB. Accesses to the K-Bus are completed in two cycles. The first cycle puts the address and attributes (read/write, length) on the bus and in the second cycle, the memory returns the data. Thus, there can be two accesses being performed in parallel on the K-Bus, one in the address phase and one in the data phase.

The MCF 5307 has a bunch of peripherals integrated, e. g. serial ports, DMA machinery, etc. Peripherals that can perform DMA are connected to the Master Bus (or M-Bus for short). The M-Bus is connected to the K-Bus via a K2M Controller. The other (slave) peripherals are connected to yet another bus, the

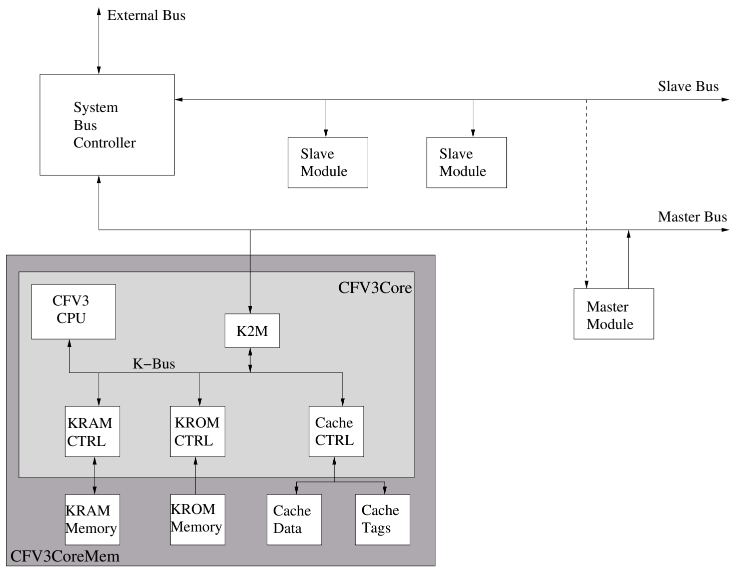


Figure 2.8: The inner architecture of the MCF 5307

slave bus which, in turn, is connected to the M-Bus. Also, the external bus of the chip is connected to the M-Bus via a System Bus Controller.

The processor core and the K-Bus run at a different clock speed than the M-Bus and the external bus. This introduces alignment delays for external accesses, since the processor core (or rather the K2M controller) has to wait until the rising edge of the external clock before starting an access.

2.5.1 The Pipeline of the ColdFire 5307

The pipeline of the MCF 5307 is depicted in Figure 2.9. It consists of a *fetch pipeline* where instructions are fetched from memory (or the cache), and an *execution pipeline* where instructions are executed. Fetch and execution pipeline are separated by a FIFO instruction buffer (IB) that can hold at most 8 instructions. The IB holds complete instructions, which can be from two to six bytes long.

The figure resembles the pipelined nature of the K-Bus in that instruction fetches are distributed over two stages (IC1 and IC2); data reads or writes are also performed in the AGEX and DSOC stages of the execution pipeline.

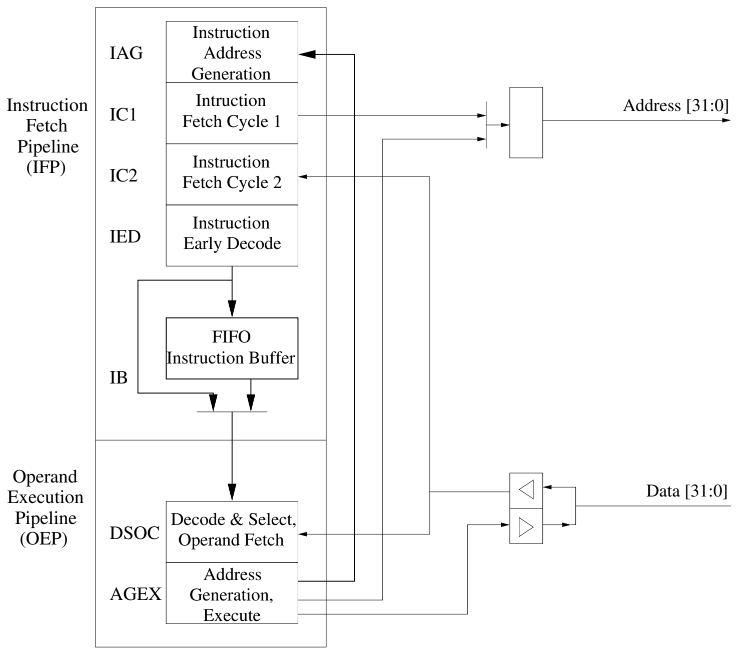

Instruction Fetch Pipeline (IFP)
IAG
Instruction Address Generation
IC1
Intruction Fetch Cycle 1
IC2
Instruction Fetch Cycle 2
IED
Instruction Early Decode
FIFO Instruction Buffer
IB
Operand Execution Pipeline (OEP)
DSOC
Decode & Select, Operand Fetch
AGEX
Address Generation, Execute
Address [31:0]
Data [31:0]

Figure 2.9: The pipeline of the MCF5307

The Fetch Pipeline

The instruction fetch pipeline of the MCF 5307 is used to generate the next fetch address, to issue the fetch via the K-Bus (and possibly via the K2M controller and further via the external bus) and to receive the fetched double words. Furthermore, it is responsible for the reassembly of instructions from the double words (four bytes) fetched.

It has 4 stages: IAG (Instruction Address Generation), IC1 (Instruction Fetch Cycle 1), IC2 (Instruction Fetch Cycle 2), and IED (Instruction Early Decode). The IED stage is responsible for performing branch prediction of conditional and unconditional branches whose target addresses are known statically, i. e., from the instruction alone.

The IAG stage generates the next fetch address. Since fetching is done in portions of one double word (four bytes), normally the next generated address is $a+4$ if a is the address currently generated by IAG. In the event that fetching has to be redirected, which can be done either by the IED stage due to branch prediction or by the AGEX stage because it discovered a misprediction or a computed call, the currently generated address is changed to the one given by either IED or AGEX.

The IC1 stage issues a fetch request via the K-Bus by placing the address generated by IAG in the previous cycle on the bus together with the appropriate attributes (instruction fetch, length). If the IC1 stage cannot issue the fetch because the bus is busy, it has to stall. In this case, it signals a stall to the IAG stage so that that stage does not generate a new address but rather keeps the current one.

The IC2 stage receives the fetched double word from the K-Bus. If it has to wait because the K-Bus cannot serve the data (external access and/or cache miss), it signals a stall to the IC1 stage (which in turn signals a stall to IAG). If the IC2 stage received the data, it forwards it to the IED stage.

The IED stage is responsible for the decoding and reassembly of instructions from the fetched double words. For this, it contains a small reassembly buffer of eight bytes. When an instruction has been reassembled, it is checked if that instruction is a branch or call that is not computed (i. e. the target address is known statically). If this is the case, the target of the instruction is predicted. For conditional branches, the target is predicted to be the target address of the taken branch, if the branch goes backward, i. e. the branch target address is smaller than the address of the branch itself. Then, the IED signals the IAG that it should generate the target address of the branch next. By this, fetching is redirected to the target of the branch. Otherwise, the branch is predicted to be not taken and no change of the fetch address is generated. If a change of the fetch address is signaled, also a signal is sent to the IC1 and IC2 stages to inform them to discard their contents. The remaining decode buffer in the IAG is cleared.

If the instruction being reassembled is a computed branch or a return instruction (RTS), then fetching is halted. The IAG, IC1 and IC2 stages are signaled to clear their contents.

The reassembled instruction is put into the IB and at the same time forwarded to the execution pipeline if the IB is empty.

In the case that either the local reassembly buffer cannot accept the next double word form IC2 or the IB is full, the IED stalls and emits a stall signal to the IC2 stage (which in turn signals a stall to IC1, etc).

Because the IB is rather large (eight complete instructions) and the IED performs static branch prediction, there may be a lot of redirections in the fetch flow, which may be canceled by the AGEX stage later (mispredictions). It is thus rather difficult to precisely analyse which memory blocks are accessed by the fetch engine.

The Execution Pipeline

The execution pipeline consists of only two stages: Decode Select and Operand fetCh (DSOC) and Address Generation and EXecution (AGEX). The DSOC finishes the decoding of the instruction and fetches the register operands. The AGEX stage executes the instruction. For certain instructions, two iterations of the instruction are required through the two stages. For instructions that perform a memory read, the AGEX stage on the first iteration generates the read address and sends it via the K-Bus. On the second iteration, the DSOC stage reads the result and the AGEX stage performs the operation (and issues the write address for a read-write kind of instruction).

By this, instructions can overlap by at most one cycle in the execution pipeline. A special case is the MOVEM instruction, which performs up to 15 read or write accesses to/from the register file. It iterates in the execution pipeline as long as there is a remaining register to be transferred.

As a side condition, a write access must be separated by at least three processor cycles from the previous write access. This does not apply to MOVEM transfer instructions.

The time it takes the instructions to execute in AGEX varies from instruction to instruction. So the AGEX stage can stall the DSOC stage.

The DSOC stage takes the first instruction from IB. If DSOC is stalled, it can happen that the IB fills completely which prefetched instructions. In this case, the IB signals this fact to the IED stage. Also, multiple iterations of one instruction through the execution pipeline do not allow a subsequent instruction to be inserted between iterations of this instruction. The next instruction can only enter DSOC if the instruction has entered AGEX in its final iteration.

Data accesses by AGEX and DSOC must use the same K-Bus as the instruction fetches from IC1 and IC2. Data accesses take precedence in the case of simultaneous accesses, i. e. the instruction fetch cannot be issued by IC1.

2.5.2 The Cache of the ColdFire 5307

The MCF 5307 features a unified instruction and data cache. The IC1 stage performs instruction accesses, the AGEX stage initiates data reads and writes. Due to the fact that instructions and data have to share the same cache, instruction prefetches may throw out data entries in the cache, leading to increased data access misses. Because the prefetching includes branch prediction along conditional branches there may be a lot of accesses to the cache, whose addresses are not easy to bound statically.

The cache of the MCF 5307 is 8kB in size, having 128 sets with four lines of 16 bytes in each set. Thus, upon an access the lower 4 bits are the index into the line of the entity being accessed (byte, word, double word), the next 7 bits are the index into the cache giving the set number and the remaining 21 bits are used as tags to search for line hits in the selected set. In addition there is one global counter of 2 bits that gives the index of the line in a set that will be replaced on the next cache miss.

If the accessed data is found in the cache (cache hit) then this counter is not updated and only the data in the line is returned via the K-Bus. If the data is not found in the cache, then it has to be loaded into the cache. Thus, an external access is started via the M-Bus and the external bus. Then the addressed set is searched for an invalid line (one containing no data). If one is found, the data read is placed into that line and also returned over the K-Bus. If all lines contain valid data, the line that is indexed by the global counter is replaced from the cache and the new data is put into that line. Then the global counter is incremented by one (wrapping around to 0 if it indexed line 3 before). This replacement strategy is called *Pseudo Round-Robin*.

Thus, accesses to different sets are not independent of one another as would be the case for other replacement strategies. This leads to strange cases, where data can stay in the cache very long.

	Way			
Set	0	1	2	3
0	0x0000	0x0800	0x1000	0x1800
1	0x0010	0x0810	0x1010	0x1810
...	...			
126	0x07E0	0x0FE0	0x17E0	0x1FE0
127	0x07F0	0x0FF0	0x17F0	0x1FF0

Table 2.2: Cache contents after 512 accesses

Consider for example that the cache is empty at a given point and the replacement counter is 0 (which is the case after a complete cache invalidation). Then we access data linearly from address 0 on, where 0 is mapped to set 0. The memory block at 0 is brought into set 0, way 0. The next block at address 16 is brought into set 1, way 0, etc. After 128 accesses the first way of all sets is filled. Access number 129 at address $0 + 128 * 16$ is mapped to set 0. Because there is an invalid line in this set, the block is put there, i. e. into set 0, way 1. This continues until we have done 512 accesses, giving the cache contents as in Table 2.2, where the addresses are given in hexadecimal notation.

After the 512th access, the replacement counter is still 0, having not been updated during all those accesses. Now access 513 is mapped again to set 0. But now we do not have an invalid line in set 0, so we must replace the line indexed by the counter, i. e. line 0. Thus, the block at address $0x2000$ is put into set 0, way 0, replacing the block at address $0x0000$. The counter is incremented by one, now indexing way 1. Access number 514 goes to set 1 and also does not find an invalid line to be placed into. Thus, the line indexed by the counter is replaced, which is line 1. This continue for the next 126 accesses, giving the cache contents of Table 2.3, where the blocks having replaced other blocks are marked in boldface.

	Way			
Set	0	1	2	3
0	**0x2000**	0x0800	0x1000	0x1800
1	0x0010	**0x2010**	0x1010	0x1810
...	...			
126	0x07E0	0x0FE0	**0x3FE0**	0x1FE0
127	0x07F0	0x0FF0	0x17F0	**0x3FF0**

Table 2.3: Cache contents after 640 accesses

Since the replacement counter was incremented 128 times and was zero (on access number 513), it is again zero after access number 640. So access number 641 to address 0x4000 again replaces the line in way 0 of set 0, as did access number 513. From here on, this scheme continues and only the ways with elements in boldface in Table 2.3 are ever replaced. The data in the other ways stays in the cache forever. This shows, that under certain access patterns only one fourth of the cache is ever being used for caching recent data. It has been observed in [HT01, HLTW03, FHL$^+$01] that indeed the analysis that can be performed models only one fourth of the cache contents, i. e. effectively a direct mapped cache of size 2kB.

The MCF 5307 has also the special feature that the cache is a wrap-around line fill, i. e. first the double word accessed in the line with a cache-miss is read over the external bus, followed by the remaining three words (wrapping around to the first word of the line after reading the last word of the line). As soon as the first word is read, it is delivered back to IC2/DSOC and the line fill continues in the background. During such a background line fill completion, the cache is able to serve accesses that do not reference the same line (hit under miss).

Things are further complicated because instruction fetches can cross cache line boundaries, necessitating two cache line accesses. In fact, the access logic splits all unaligned accesses into up to three aligned accesses on the K-Bus.

The cache semantics (cf. Section 2.1) for the ColdFire cache is complicated by the fact that sets in the cache are not independent. Updating one set changes the ages in the other sets, as there is only one global replacement counter. The age is in this case defined as follows: the line that is pointed to by the replacement counter has age 3. The next line has age 2 and so on, wrapping around after line 3 to line 0.

Cache hits do not change the replacement counter, as do cache misses if there is an invalid line in the set referenced by the access. All other misses replace the line pointed to by the replacement counter and increase the counter by one (modulo 4).

The set of values of the replacement counter is $R = \{0,1,2,3\}$. We model the cache c as a pair of replacement counter and a *set mapping*. A set mapping s maps a set index to a *line mapping*, which maps the line index to the memory block contained in that line or $\top$ if the line is invalid:

$$\begin{aligned} c &: [c], [c] = R \times (\{0,\ldots,127\} \rightarrow (\{0,\ldots,3\} \rightarrow M^\top)) \\ s &: [s], [s] = \{0,\ldots,3\} \rightarrow M^\top \end{aligned}$$

The function $\text{loc} : [c] \times M \rightarrow \{0,1,2,3,\top\}$ maps a memory block to the line it is contained in in the cache, or $\top$ if it is not in the cache:

$$\text{loc}((r,d),m) = \begin{cases} l & \text{, if } \exists l : m = d(\text{set}(m))(l) \\ \top & \text{, otherwise} \end{cases}$$

The update of the cache by an access to memory block m is described by the functions $\mathcal{U}^c : [c] \times M \rightarrow [c]$:

$$\mathcal{U}^c((r,d),m) = \begin{cases} (r,d) & \text{, if } \text{loc}((r,d),m) \neq \top \\ (r', d[\text{set}(m) \mapsto s']) & \text{, otherwise} \end{cases}$$

where $(r',s') = \mathcal{U}^s(d(\text{set}(m)), r, m)$.

The function that updates one set and the replacement counter is $\mathcal{U}^s : [s] \times R \times M \rightarrow R \times [s]$:

$$\mathcal{U}^s(s,r,m) = \begin{cases} (r, s[l \mapsto m]) & \text{, if } \exists l' : s(l') = \top, l \text{ smallest such } l' \\ ((r+1) \bmod 4, s[r \mapsto m]) & \text{, otherwise} \end{cases}$$

To classify a memory access as a cache hit H or cache miss M, the function $\text{classify} : [c] \times M \rightarrow \{H, M\}$ is defined:

$$\text{classify}(c,m) = \begin{cases} H & \text{, if } \text{loc}(c,m) \neq \top \\ M & \text{, otherwise} \end{cases}$$

2.5.3 System Configuration

The MCF 5307 allows to map the internal SRAM area at different addresses, controlled by a system register. Also, the memory area that the registers of the integrated peripherals are mapped to can be chosen by setting an internal register with the `movec` instruction.

Furthermore, two access registers and a cache control register are used to configure access properties for memory areas (i. e. up to three memory areas with different properties can be configured). Configurable properties include caching mode (uncached, cached write-through, cached write-back) and the use of an internal burst buffer to accelerate memory reaccesses.

Furthermore, memory controllers are built into the MCF 5307 for DRAM or SDRAM and asynchronous access using up to 8 chip-select signals. The memory areas corresponding to these external accesses are configurable by registers in the peripheral register area.

2.5.4 Assumptions Made

Naturally, it is not possible or necessary to model all features of a processor for an analysis. By placing certain restrictions on the code to be analyzed, the analyzer can be simplified and made more efficient.

Mostly, the features that can cause problems are not used in the area of real-time programming anyhow, while others are easy to avoid. For our analyzer, we made a number of restrictions on the underlying hardware of the system, the programming of system control registers and the behavior of programs.

Underlying Hardware

We assume that the system only uses SRAM or/and EEPROM for main memory and that *all* access times over the external bus are fixed, either to SRAM or to peripherals or to EEPROM (Flash). No other bus masters must use the external bus and the internal DMA engines of the MCF 5307 must not be used. Thus, the timing parameters for memory accesses are determined by the address of the access alone. Thus, bus contention or variable access times need not be modeled.

System Configuration

We assume that the program (or rather, a part of the program whose timing is of interest) does not change any settings in system registers. That is, the memory area configuration is once initialized at system startup (whose timing is not analyzed) and then never changed. We assume that all cacheable memory areas are configured for write-through mode and that the burst buffer is not used on uncacheable memory areas. This allows us to ignore the side effects of setting system registers in the model. Write-back mode is impossible to analyze precisely due to the restricted form of cache analysis possible for the MCF 5307.

The cache is only modified by memory accesses made by the program (code or data) and not manipulated by the special `cpushl` instruction. Cache locking (an option offered by the MCF 5307) is not used. The effects of the `cpushl` instruction are difficult to model precisely because the cache analysis cannot find guaranteed cache misses. The possible memory accesses by this instruction would make the results very imprecise. Cache locking would lead to imprecision for the same reasons.

Programming

We assume that the program runs without causing exceptions and with interrupts turned off. Instructions that manipulate CPU internal system registers (`movec`) must not be used. Furthermore, instructions that cause special behavior, e. g. the `stop` instruction must not be used (this instruction halts program execution until an interrupt occurs). Other special instructions are “move to SR”, `pulse`, `trap`, `wddebug` and `halt`. Disallowing these instructions enables us to simplify the model considerably, since we do not have to model their complex behavior.

Furthermore, no self-modifying code is allowed and no dynamic allocation (at the level of the OS). The code in the system must conform to the ABI[2] set forth by Motorola for the ColdFire family (e. g. stack pointers must resume the same value after a call to a function that they had before the call). This allows to ignore data accesses w.r.t. the form of the control-flow graph of the program.

For memory areas mapped to read-only devices, no write accesses may occur. If a memory has alignment restraints for accesses, the program must not access it misaligned (e. g. some control register must not be accessed at odd memory addresses or can only be read by a word-sized transfer).

For data accesses, data must only be accessed naturally aligned in main memory. E.g. an access of a 16bit word must only occur at an even address.

Thus, we can ignore the special behavior caused by misaligned accesses in the modeling process.

2.5.5 Timing Anomalies with the MCF 5307

Even though the MCF 5307 pipeline is a rather simple in-order architecture, it shows timing anomalies. One anomaly steams from the replacement strategy of the cache, which falsifies one commonly made local worst-case assumption: on the MCF 5307 an empty cache is not the worst-case for program execution.

Another anomaly originates in the fact that fetch and execution pipeline are independent (only coupled by the IB and prediction resolution) but use the same unified cache.

[2]Application Binary Interface: defines e. g. the layout of stack frames and calling conventions of procedures.

Empty Cache not Worst-Case

The assumption that an empty cache constitutes the worst possible cache w.r.t. execution time behavior (as there cannot be any cache hits from this cache contents), is not true for the MCF 5307.

Due to the dependence of the cache sets of the MCF 5307 on the 2-bit replacement counter, some cache configurations and access patterns can lead to the effect that newly loaded blocks throw out blocks loaded into a set on the last access to that set, cf. Table 2.3. If the cache is empty, the first four accesses to a set will place new data into the set, and replacement can only happen on the fifth access to that set.

Table 2.4 shows an access sequence of 8 accesses to sets 0...3 of the cache. The accesses go to addresses 0x0, 0x10, 0x20, 0x30, 0x800, 0x810, 0x820, 0x830, so sets 0...3 each are accessed twice by this sequence.

	Empty Cache				Filled Cache			
Set	Line				Line			
	0	1	2	3	0	1	2	3
After 4 accesses								
0	**0**	I	I	I	**0**	1800	2000	2800
1	**10**	I	I	I	1010	**10**	2010	2810
2	**20**	I	I	I	1020	1820	**20**	2820
3	**30**	I	I	I	1030	1830	2030	**30**
After 8 accesses								
0	0	**800**	I	I	**800**	1800	2000	2800
1	10	**810**	I	I	1010	**810**	2010	2810
2	20	**820**	I	I	1020	1820	**820**	2820
3	30	**830**	I	I	1030	1830	2030	**830**

Table 2.4: Timing anomaly MCF 5307: Empty cache not worst-case (first 8 accesses)

The table shows cache misses as **boldface** and cache hits in a set in *italics*. Assume that the non-empty cache is filled with memory blocks 0x1000, 0x1800, 0x2000, 0x2800 for set 0 and blocks 0x1010, 0x1810, 0x2010, 0x2810 for set 1, etc, and the replacement counter is 0 before the first access.

For the empty start cache, after 8 accesses the eight memory blocks are in ways 0 and 1 of sets 0...3, causing 8 cache misses. For the other cache, after four accesses, the first four memory blocks are in the sets, and the replacement counter is again 0. So the next four accesses to blocks 0x800,... replace the blocks just loaded by the previous accesses. After eight accesses, only the last four blocks of the sequence are in the cache and there have been eight cache misses.

	Empty Cache				Filled Cache			
Set	Line				Line			
	0	1	2	3	0	1	2	3
After 12 accesses								
0	*0*	800	I	I	**0**	1800	2000	2800
1	*10*	810	I	I	1010	**10**	2010	2810
2	*20*	820	I	I	1020	1820	**20**	2820
3	*30*	830	I	I	1030	1830	2030	**30**
After 16 accesses								
0	0	*800*	I	I	**800**	1800	2000	2800
1	10	*810*	I	I	1010	**810**	2010	2810
2	20	*820*	I	I	1020	1820	**820**	2820
3	30	*830*	I	I	1030	1830	2030	**830**

Table 2.5: Timing anomaly MCF 5307: Empty cache not worst-case (16 accesses)

Now assume that this access sequence is repeated, which gives the results shown in Table 2.5. For the empty start cache, all accesses are cache hits, since the blocks have been previously loaded into the cache. For the filled cache, every access is a cache miss, because the subsequences of four accesses throw out the blocks needed by the next sequence of 4 accesses. After 16 accesses, the empty cache has lead to 8 cache misses to load the 8 blocks and 8 cache hits due to block reuse. The filled cache has lead to 16 cache misses. In general, for $8n$ accesses with this sequence, the empty cache leads to 8 cache misses and $8(n-1)$ cache hits, while the other cache results in $8n$ cache misses. As the instructions performing the accesses and the loop counter increment and test also take time to execute, n iterations of the loop take $8M + 8(n-1)H + nO$ cycles (empty case), and $8nM + nO$ cycles (filled) cache, where M is the miss penalty, H the access time for a cache hit and O the remaining overhead of the instructions in the loop.

In Figure 2.10 an example program for this anomaly is depicted, the measurements of the execution times are depicted in Figure 2.11 for the empty and filled

```
lea.l     0x30000,a2
move.l    #N,D3
Loop:
move.l    0x00(A2),D1
move.l    0x10(A2),D1
move.l    0x20(A2),D1
move.l    0x30(A2),D1
move.l    0x800(A2),D1
move.l    0x810(A2),D1
move.l    0x820(A2),D1
move.l    0x830(A2),D1
subq      #1,D3
bne       Loop
```

Figure 2.10: Timing anomaly MCF 5307: Program

cache scenarios. The loop in the program is executed N times. The number of cycles is measured for the code sequence starting from the label *Loop* until after the final branch of the last iteration[3]. The behavior of a filled cache is equivalent to no caching at all. Turning off the cache and running the example program indeed gives the same timing results as for the filled cache.

As an empty cache is often assumed to be the worst-case scenario for program execution, this behavior is of substantial impact for WCET computation. Unfortunately, it is not obvious *which* concrete cache will be the worst-case for a program execution, as this depends on the memory access sequence of the program. Thus, more abstract notions of "undetermined cache" must be used as descriptions of cache contents at program start. The theory of abstract interpretation underlying our WCET tool readily provides a solution for this problem.

Unified Cache and Prefetching

The combination of prefetching and a unified cache leads to a scenario, where a data access of one instruction that is a cache miss leads to a shorter global

[3]The measurements were made on an MCF 5307 Evaluation board running at 45/90MHz (bus/core) using the built-in timers of the MCF5307. The program code was in the internal SRAM so it does not influence the cache.

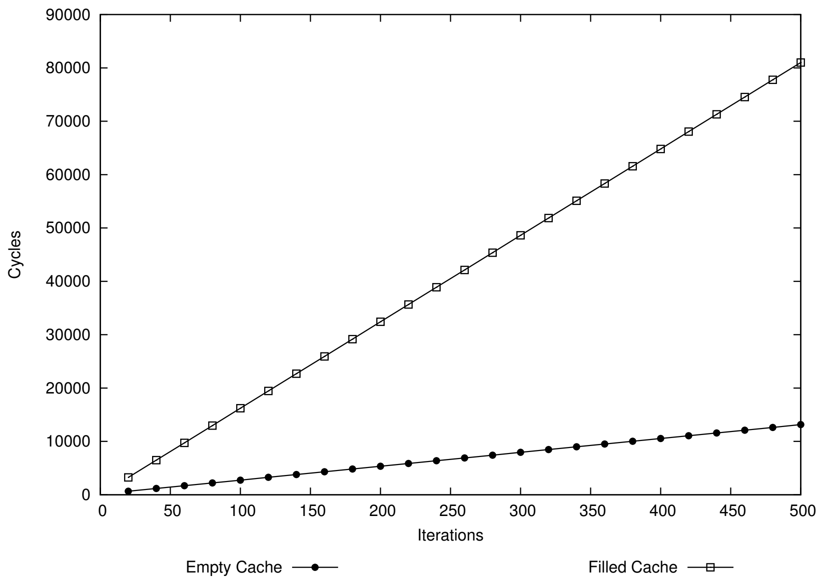


Figure 2.11: Timing anomaly MCF 5307: Measurements

execution time of a complete program sequence compared to a cache hit by that instruction's data access.

This happens because in the case of the data access being a cache hit, the prefetching of the IFP can continue, resulting in a cache miss that replaces *two* lines, because of an instruction crossing a cache line boundary. Assuming that those two lines contain useful data, successive data accesses to those lines result in two additional cache misses. If the branch was mispredicted, those two lines are fetched without being used in program execution.

If the data access is a cache miss, then prefetching cannot continue, as the memory bus is in use by the data access. After the data access finishes, the execution pipeline can resolve the branch condition before the misprediction of the branch has led to a fetch of the two cache lines. In this case, only one cache miss occurs, compared to two misses in the first scenario.

2.6 The Motorola PowerPC 755

2.6.1 The PowerPC Architecture

The architecture of the PowerPC family of microprocessors covers 32-bit and 64-bit variants. The 32-bit architecture is described in [PPC97b], which fixes the set of required and optional features of an implementation. The PPC 755 is a 32-bit PowerPC.

For our purposes the following elements of the architecture are important:

- A set of 32 32-bit *General Purpose Registers* (GPR)
- A set of 32 64-bit *Floating Point Registers* (FPR)
- A set of *Special Purpose Registers* (SPR) of (at most) 32-bit each
- Some special registers: the link register (LR), the count register (CTR), the condition code register (CCR)
- Some instructions provide instruction synchronization and/or context synchronization
- Memory access translation: with IBAT/DBAT registers.

2.6.2 The PowerPC 755

The PPC 755 is an improved variant of the PPC 750, which differs from the latter CPU only by an enhanced L2-Cache interface, some additional SPRs, an advanced method to lock the instruction and/or data cache, support for software TLB searches and a selectable reduced bus interface. Since we are assuming that most of these features will not be used in the software considered, we can take the PPC 750 manual, [PPC97a] as the basis of our description and model[4] [5].

2.6.3 PPC 755 Pipeline

The pipeline of the PowerPC 755 is depicted in Figure 2.12 and contains the following units:

[4]We will sometimes directly reference to chapters in both the architecture and PPC 750 manual in the form [PPC97a, *Chapter*] etc.

[5]The only PPC 755 features that are considered in the model are certain cache locking configurations and instructions, like `dcbf`.

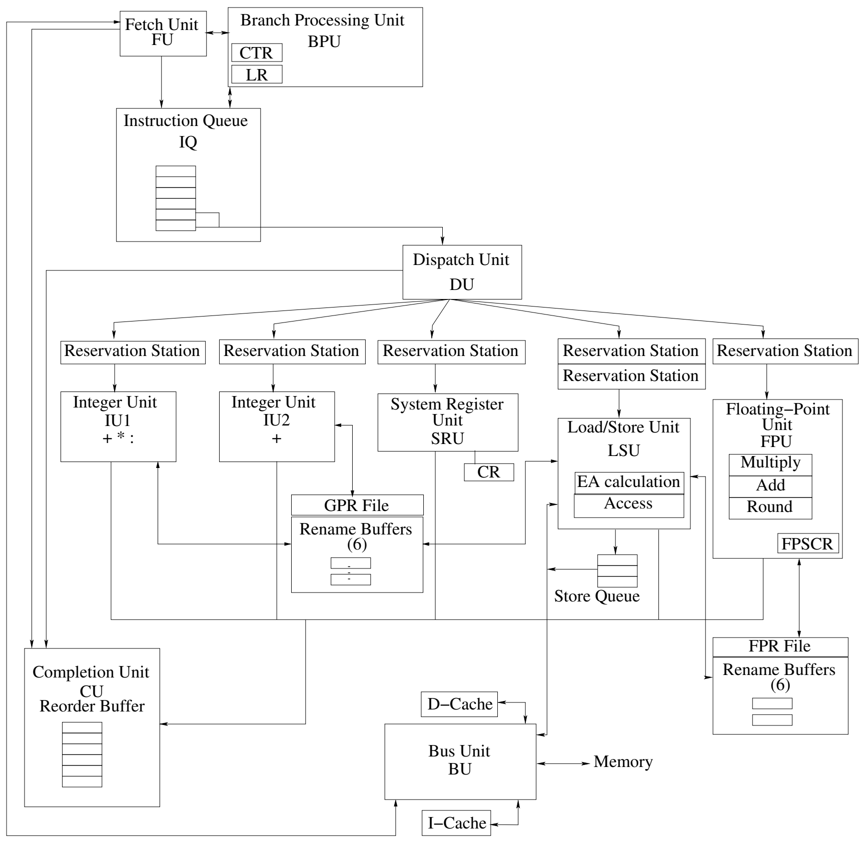


Figure 2.12: The pipeline of the PowerPC 755

- The *Fetch Unit* (FU) fetches one to four instructions from memory (possibly via the instruction cache) and places them into the *Instruction Queue* (IQ). How many instructions are requested depends on the number of free slots in IQ: the requested number is the minimum of 4 and the number of free slots in the IQ.

- The Instruction Queue (IQ) holds instructions to be dispatched. It is organized as a FIFO with six entries.

- The *Branch Processing Unit* (BPU) takes care of predicting and performing branches. It reads the instructions as soon as they are in the IQ, removes taken (or predicted taken) branches, and redirects the instruction fetch accordingly. The BPU contains a branch target instruction cache and a branch history table to dynamically predict branches. Since we will not use this feature, these components are not shown in the figure.

- The *Dispatch Unit* (DU) dispatches instructions from IQ0 and IQ1 to the five functional units. Instructions are dispatched in-order, i.e. an instruction can only be dispatched from IQ1, if the instruction in IQ0 is also dispatched.
- The *Completion Unit* (CU) contains six entries that are assigned to dispatched instructions (except for some branch instructions handled directly by the BPU). The retirement of the instructions happens from the CU in order, i.e. an instruction can only retire if its predecessors are already retired. The PPC 755 can retire at most two instructions from the CU per cycle.
- The *Integer Units* (IU1/IU2) perform integer operations. IU1 can perform all operations, while IU2 cannot perform multiplications and divisions.
- The *System Register Unit* (SRU) handles operations on the SPRs, like e. g. `mfspr`.
- The *Load/Store Unit* (LSU) performs loads to GPRs/FPRs from memory and write operations from those registers to memory. It is pipelined with two stages: the first stage performs the calculation of the effective address of the operation and the second stage performs the memory access. Stores from the LSU are first gathered in a *Store Buffer* of three entries. This is not true for multiple store instructions like **stmw** which perform stores directly.
- The *Floating-Point Unit* (FPU) handles all floating-point operations and consists of a pipeline with three stages: the first stage performs multiplications and divisions. The second stage performs additions and the last stage rounds or converts the result.
- The GPR file and FPR file contain 32 registers each and in addition six integer rename registers and six floating-point rename registers.
- The *Bus Unit* (BU) is responsible for all accesses via the external bus. Data requested from the FU or the LSU (or written from the Store Queue) may also go through the instruction or data cache.

Instruction execution is handled in the following manner: first, instructions are fetched by the FU, possibly via the I-Cache, from memory. The number of instructions fetched depends on the number of free slots in the IQ. If in cycle n there are m free slots in the IQ, then $\min(4,m)$ instructions are requested by the fetcher. Even if in cycle n one to three slots may be freed by instruction dispatch and branch processing, these slots are not counted in the number of free slots in that cycle ([PPC97a, 6.3.1]). The instructions are put into the IQ in program order, i.e. in address order.

Instructions in the lowest two entries in IQ (IQ0 and IQ1) can be dispatched for execution. Dispatch happens in-order, so the instruction in IQ1 can only be dispatched, if the instruction in IQ0[6] can be dispatched. Dispatching from IQ0 is restricted by several rules ([PPC97a, 6.6.1.2]):

- The execution unit needed by [IQ0] is available
- Needed GPR and/or FPR rename buffers are available
- An entry in the CU is available
- No completion serialization instruction is being executed (see page 63)

Dispatching from IQ1 can only happen, if

- [IQ0] can be dispatched
- [IQ0] is not a completion or refetch serialization instruction (see page 63)
- Execution unit needed by [IQ1] is available (with [IQ0] already dispatched)
- GPR/FPR rename buffer available (after [IQ0] dispatched)
- CU not full ([IQ0] already dispatched)

For each instruction an entry in the CU is allocated in dispatch (i. e. program) order. Certain instructions don't take an entry in the CU and are removed from the instruction flow by the BPU; these instructions are branch instructions that do not update the LR or CTR.

Also, some branch instructions have additional requirements (cf. [PPC97a, 6.6.1.1]):

- The `bclr` requires that the LR is available, i. e. there is no outstanding computation writing the LR.
- The `bcctr` requires that CTR is available.
- Branch and link instructions require that a rename LR register is available.
- The 'branch conditional on ctr decrement' instruction requires CTR availability (if the condition is not false).

[6]We write [IQ*n*] to denote the instruction in IQ*n*.

- A branch conditional instruction cannot be executed following an unresolved branch. I.e. there can be at most one level of speculative execution.

Branches are also treated specially, when they enter the IQ:

- As soon as a branch is fetched, it is examined to decide whether it can be folded or fallen-through (i. e. removed from the instruction stream).
- Instructions are treated speculatively if the condition on which they depend is not known. This is called an unresolved branch.

If a branch instruction that is unconditional and whose operands are available enters the IQ, it is folded away if it is taken. The branch and all later instructions in IQ are spilled and instruction fetch continues at the target of the branch. If the branch is not taken, it remains in IQ until it reaches IQ0 or IQ1 and is then simply discarded ([PPC97a, 6.4.1.1]).

Conditional branch instructions are treated like unconditional branches if the condition is already known (i.e. the CR will not be modified by any instruction in the CU or IQ). If this is not the case they are predicted according to their instruction encoding (cf. [PPC97b, 4.2.4.2]). If they are predicted as not taken, they will fall-through. Otherwise, they will be folded. The processor will mark all subsequent instructions as speculative until the condition is resolved. If the condition turns out to be different than predicted, all instructions after the branch are flushed from CU and the functional units, and instruction fetch continues along the other branch path.

When an instruction can be dispatched, it allocates the required rename registers[7]. After that, the instruction is assigned an entry in CU and it is dispatched to the reservation stage of the functional unit required by the instruction[8]. If the operands required by that instruction are not available, it will stay in the reservation stage until the instruction(s) that compute those operands are finished and broadcast their results.

The LSU and FPU are pipelined, so two instructions can be in the LSU at the same time or three instructions in the FPU. Some instructions cannot be pipelined (e.g. certain FPU instructions like `fdiv`, cf. [PPC97a, 6.4.3]) and block the whole unit until they are finished. Instructions already in the later stages of the FPU continue their execution concurrently with a blocking instruction.

The LSU unit has two stages: EA calculation and access. If an instruction in the LSU is speculative (after an unresolved branch), then it must not access

[7]For instructions that update CR, LR and CTR there is one rename register for each of those registers.

[8]For integer instructions, one of IU1 and IU2 can be chosen, if both are free and the instruction doesn't perform multiplication or division. We assume that IU2 is chosen to keep IU1 free for multiplications in this case.

memory marked as ‘guarded’ in the DBAT register for that access. This means, such an access will stall in the access stage until the branch is resolved (until the speculative bit in the CU entry for that instruction is cleared). Stores are only performed, when the instruction is retired from the CU, except for multiple store instructions, like **stmw**, which are completion serialization instructions and thus can never be executed speculatively. These instructions perform stores directly. All other stores are kept in a three-entry store buffer at the LSU and are performed out of that buffer when the instruction is retired from the CU. If it must be flushed due to a misprediction, the store is discarded from the store buffer.

We assume that a load reads data from the store buffer if it accesses the same data as a previous store which is still not committed.

Completion of instructions can only happen from CU0 and CU1, restricted by ([PPC97a, 6.6.1.3]):

- [CU0] must be finished
- [CU0] must not follow an unresolved branch
- [CU0] must not cause an exception

For [CU1], similar rules apply:

- [CU0] must complete in the same cycle
- [CU1] must be finished and must not follow an unresolved branch
- [CU1] must not cause an exception
- [CU1] *must be an integer or load instruction.*
- there must not be more than two CR updates from both [CU0] and [CU1]
- there must not be more than two GPR updates[9] for [CU0] and [CU1].
- no more than two FPR updates may occur

Certain instructions require *synchronization*. Synchronization or serialization can happen in three ways ([PPC97a, 6.3.3.2]):

- *Execution Serialization*: These instructions are dispatched to a unit but do not begin execution until all previously issued instructions have completed. Example: `mtspr`.

[9]So we cannot complete a load with update instruction and an addition in the same cycle.

- *Completion Serialization*: An instruction does not start execution until all previously issued instructions have completed. In contrast to the execution serialization, no instructions after this instruction may be issued to functional units until it has completed execution. Example: `lswi`.

- *Refetch Serialization*: Behaves like completion serialization but in addition, all instructions in the IQ after this one are flushed and have to be refetched when the instruction completes. Example: `isync`.

2.6.4 PPC 755 Caches

The PPC 755 has two separate caches for instructions and data. Internally, the data paths to these caches are independent (Harvard architecture). Each Cache has a capacity of 32kB and is eight-way set associative with a line size of 32 bytes (4 double words of 64 bits), thus each cache has 128 ways. The replacement strategy of the caches is called *pseudo least recently used* (PLRU) by Motorola. Again, as in the case of the pseudo round robin scheme of the Coldfire, the "pseudo" leads to a loss in predictability. When accessing an address, instruction or data, in an area that is marked as cacheable by the corresponding BAT registers, bits $27-31$ select the byte in the line[10]. Bits $20-26$ select the set in the cache, bits $0-25$ are compared against the tags of the (up to) eight lines in that way. Each line in the set has a valid bit that is set if the line contains a data block. For the data cache, another bit, the *dirty bit*, signals if the line has been written to and must be copied to main memory on cache flush or replacement.

If a cache hit occurs, the data is delivered in the following cycle. For a cache miss, new data is loaded into the set. If there is an invalid line in the set, the newly read line is placed in that line. If there are multiple invalid lines in the set, the one with the lowest index $(0 \dots 7)$ is chosen. If there is no invalid line, i. e. the set is completely full, one line is chosen to be replaced. A replacement in the data cache may invoke a write-back of the line replaced, if it is dirty. Which line to choose for replacement is governed by a logic that uses 7 state bits. Figure 2.13 depicts the behavior of this selection logic.

In this figure, the 8 lines in a set are at the bottom and the seven state bits are represented by the circles. Starting with bit 0 at the top, a path to a line is chosen according to the state, 0 or 1, of each bit. E.g. if the 7 seven bits are (with bit 0 at the left) 0110011 then line 2 is chosen. For each set, there is a separate set of these seven bits, so unlike the ColdFire cache, set accesses are independent of one another.

[10]Motorola counts the bits in a 32 bit word for the PowerPC in such a way that bit 0 is the *most significant bit* with value 2^{31} and bit 31 the least significant with value 2^0. Likewise for 64 bit double words.

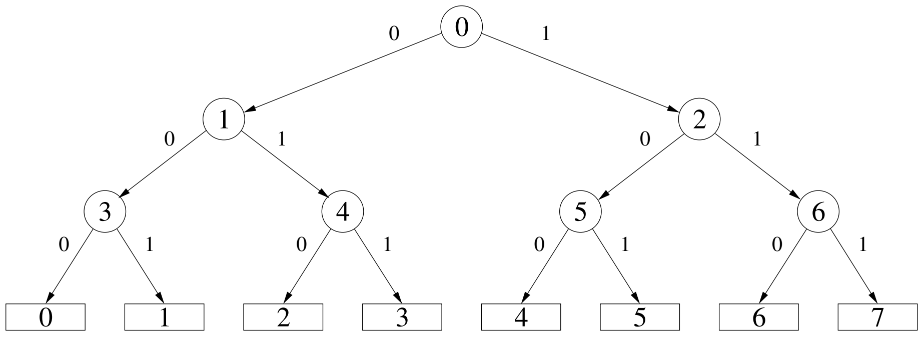


Figure 2.13: PowerPC 755 PLRU replacement selection logic

After a hit or a miss in a cache set, the state bits for that set are updated. This update depends on the line in the set that has been accessed. Intuitively, the bits in all three layers are changed so that the next line selected by the new setting points away as far as possible from the old line. This means that every bit on the path from bit 0 to the accessed line is inverted. Table 2.6 gives the update function on the state bits. It denotes for the accessed line the new settings of the state bits, where "U" means that the corresponding bit is unchanged.

	New State Bits						
Line	0	1	2	3	4	5	6
0	1	1	U	1	U	U	U
1	1	1	U	0	U	U	U
2	1	0	U	U	1	U	U
3	1	0	U	U	0	U	U
4	0	U	1	U	U	1	U
5	0	U	1	U	U	0	U
6	0	U	0	U	U	U	1
7	0	U	0	U	U	U	0

Table 2.6: PPC 755 state bits update

This replacement scheme also has a pathologic access sequence that lets some part of the cache contents survive although it is not reused. One sequence that shows this behavior is an alternation of accesses that hit in line 0 of the set and then miss and replace in the line pointed to by the state bits. Assuming that all state bits are zero initially and that valid data is in all lines, Table 2.7 gives an access sequence. Here, the first column gives the running number of the sequence, the second shows the line that is accessed, which is either line 0 for a cache hit (odd numbered accesses) or the line pointed to by the state bits for a cache miss and replacement. The third column presents the state bits after the access, the last column gives the line pointed to by the state bits from column three. The even numbered accesses (in **bold**) are misses that replace data in the lines pointed to by the state bits from the preceding row in the table.

#	Accessed	State bits	Pointer
1	0	1101000	4
2	**4**	**0111010**	**2**
3	0	1111010	6
4	**6**	**0101011**	**2**
5	0	1101011	5
6	**5**	**0111001**	**2**
7	0	1111001	7
8	**7**	**0101000**	**2**
9	0	1101000	4

Table 2.7: PPC 755 bad replacement sequence

This sequence ends with the same state bits as after the first access, so this scheme of lines replaced will repeat itself. Table 2.8 shows the cache contents of the set after each of the accesses. The access sequence starts at address 0 and shows which memory blocks (in hexadecimal notation) are in the set after the access. The lines affected by the access are in bold. Lines affected by previous accesses are in *italics*. The access sequence is 0, 0x8000, 0, 0x9000, 0, 0xa000,. . .

Only lines 4, 6, 5 and 7 are ever replaced, while lines 1, 2 and 3 stay in the cache (although untouched). This means that those three lines not in italics or bold in the last row of Table 2.8 do not take part in the replacement scheme for this access pattern.

Even with a sequence of accesses where each access address is distinct from the others it may take up to 16 misses and a total of 21 accesses until all lines in a cache set have been replaced, as exemplified in Table 2.9. There, the accesses are in bold and *new* lines are denoted in italics. In this example, line 7 survives until access 21. With a real LRU strategy a line would survive at most 15 unique accesses.

However, if one looks at a sequence of accesses where each access results in a cache miss, then for every setting of the state bits at the beginning of the sequence, every line survives at most 7 accesses. That is, all lines are replaced after 8 accesses.

The precise cache semantics is defined with a cache $c \in [c]$ as a (total) mapping from set indices to a pair of line mapping and replacement bits. The line mapping $s \in [s]$ maps line indices to the memory block contained in that line or $\top$ if the line is invalid:

$$\begin{aligned}
&[c] = \{0,\dots,127\} \to (\{0,\dots,7\} \to M^{\top}) \times R \\
&[s] = (\{0,\dots,7\} \to M^{\top}) \times R \\
&R = \{0,\dots,127\}
\end{aligned}$$

The update of a cache c after a memory access to block m uses the location of a memory block in a set (its line), which is given by two functions, $\mathrm{loc} : [c] \times M \to$

#	Line							
	0	1	2	3	4	5	6	7
1	**0000**	4000	2000	6000	1000	5000	3000	7000
2	*0000*	4000	2000	6000	**8000**	5000	3000	7000
3	**0000**	4000	2000	6000	*8000*	5000	3000	7000
4	*0000*	4000	2000	6000	*8000*	5000	**9000**	7000
5	**0000**	4000	2000	6000	*8000*	5000	*9000*	7000
6	*0000*	4000	2000	6000	*8000*	**a000**	*9000*	7000
7	**0000**	4000	2000	6000	*8000*	*a000*	*9000*	7000
8	*0000*	4000	2000	6000	*8000*	*a000*	*9000*	**b000**
9	**0000**	4000	2000	6000	*8000*	*a000*	*9000*	*b000*
10	*0000*	4000	2000	6000	**c000**	*a000*	*9000*	*b000*

Table 2.8: PPC 755 cache set contents

$\{0,\ldots,7,\top\}$, and $\text{loc}^s : [s] \times M \rightarrow \{0,\ldots,7,\top\}$:

$$\text{loc}(c,m) = \text{loc}^s(c(\text{set}(m)),m)$$

$$\text{loc}^s((s,r),m) = \begin{cases} l & \text{, if } \exists l : m = s(l) \\ \top & \text{, otherwise} \end{cases}$$

The result $\top$ means that the memory block is not in the cache.

As the sets of the cache are independent the cache update is simply the application of the set update function to the correct set:

$$\mathcal{U}(c,m) = c[\text{set}(m) \mapsto \mathcal{U}^s(c(\text{set}(m)),m)]$$

	Line							
#	0	1	2	3	4	5	6	7
1	**09000**	02000	03000	04000	05000	06000	07000	08000
2	*09000*	02000	**03000**	04000	05000	06000	07000	08000
3	*09000*	**02000**	03000	04000	05000	06000	07000	08000
4	*09000*	02000	03000	**04000**	05000	06000	07000	08000
5	*09000*	02000	03000	04000	**0a000**	06000	07000	08000
6	**0b000**	02000	03000	04000	*0a000*	06000	07000	08000
7	*0b000*	02000	03000	04000	*0a000*	**06000**	07000	08000
8	*0b000*	02000	**0c000**	04000	*0a000*	06000	07000	08000
9	*0b000*	02000	*0c000*	04000	*0a000*	06000	**0d000**	08000
10	*0b000*	**0e000**	*0c000*	04000	*0a000*	06000	*0d000*	08000
11	*0b000*	*0e000*	*0c000*	04000	**0f000**	06000	*0d000*	08000
12	*0b000*	*0e000*	*0c000*	**10000**	*0f000*	06000	*0d000*	08000
13	*0b000*	*0e000*	*0c000*	*10000*	*0f000*	06000	*0d000*	**08000**
14	**11000**	*0e000*	*0c000*	*10000*	*0f000*	06000	*0d000*	08000
15	*11000*	*0e000*	*0c000*	*10000*	*0f000*	**12000**	*0d000*	08000
16	*11000*	*0e000*	**13000**	*10000*	*0f000*	*12000*	*0d000*	08000
17	*11000*	*0e000*	*13000*	*10000*	*0f000*	*12000*	**14000**	08000
18	*11000*	**15000**	*13000*	*10000*	*0f000*	*12000*	*14000*	08000
19	*11000*	*15000*	*13000*	*10000*	**16000**	*12000*	*14000*	08000
20	*11000*	*15000*	*13000*	**17000**	*16000*	*12000*	*14000*	08000
21	*11000*	*15000*	*13000*	*17000*	*16000*	*12000*	*14000*	**18000**

Table 2.9: PPC 755 unique access sequence

If a set contains an invalid line and a cache miss occurs, that line is filled with the new memory block (lowest line first). Otherwise, the line selected by the replacement bits is replaced. After every access, the replacement bits are updated according to Table 2.6. Let $\text{up} : R \times \{0, \ldots, 7\} \to R$ be this update function and let $\text{line} : R \to \{0, \ldots, 7\}$ be the function mapping replacement bits to the line selected by them, according to Figure 2.13.

Then the set update $\mathcal{U}^s : [s] \times M \to [s]$ is defined as

$$\mathcal{U}^s((s,r),m) = \begin{cases} (s, \text{up}(r, \text{loc}^s(s,m))) & \text{, if } \text{loc}^s(s,m) \neq \top \\ (s[l \mapsto m], \text{up}(r,l)) & \text{, if } \ \text{loc}^s(s,m) = \top \wedge \\ & \qquad \exists l : s(l) = \top \wedge l \text{ min.} \\ (s[\text{line}(r) \mapsto m], \text{up}(r, \text{line}(r))) & \text{, otherwise} \end{cases}$$

To determine if a cache access is a cache hit H or cache miss M, the function $\text{classify} : [c] \times M \to \{H, M\}$ is used:

$$\text{classify}(c,m) = \begin{cases} H & \text{, if } \text{loc}(c,m) \neq \top \\ M & \text{, otherwise} \end{cases}$$

2.6.5 Assumptions Made

The PPC 755 has some features with great impact on the desired pipeline model. Some of these features can be controlled by configuration registers, others can be avoided by following a certain style of programming. Again, it is desirable to exclude certain features or behaviors in order to reduce the complexity of the model.

We made the following assumptions in the design of our pipeline analyzer:

- The PPC 755 can be used in real address mode, i.e. without address translation for data and/or instruction accesses. Since this mode poses some strange conditions with regard to the 'guarded' attribute of memory (cf. [PPC97b, 5.1.2.5.3]) and speculative execution, we will assume that always only the DBAT/IBAT registers are used to perform block address translation. This assumption allows us to drop the modeling of virtual memory, which is complicated and not possible precisely. Also, we can reduce the complexity of the memory access model in the LSU.

- The PPC 755 can perform *Store Gathering* ([PPC97a, 2.3.4.3.5]). We do not model this feature, which can be turned off through the HID0 SPR. Allowing store gathering increases complexity and reduced precision of the analysis as we will not be able to precisely determine that stores can be gathered for imprecise data accesses.

- All synchronous exceptions on the 755 are precise. However, we will not model any exceptions, including floating-point exceptions or illegal instructions. Exceptions are mostly a sign of an error that occured during program execution. If an error occurs, the system will go to recovery mode or shut down. Either way, no WCET is needed for this case and we can ignore it in our timing model.

- The 755 can use an external L2 cache. We do not model that cache and assume that it is turned off. Also, we will not model the case that this L2 cache can be configured as a private SRAM area. Access to the L2 is complicated and increases the complexity of the interface to the caches. The L2 hardware was not present in the system the analysis was originaly designed for anyhow.

- We do model only a subset of the data cache invalidate or flush instructions, the others are assumed not to be present in the program. This is because the semantics of these instructions is difficult and their behavior cannot be analyzed precisely later.

- Dynamic branch prediction with a branch target instruction cache (BTIC) and a branch history table (BHT) is built into the 755. We do not model these features, and assume that they are turned off in HID0. Dynamic branch prediction can in principle be modeled with increased complexity. However, it does not give a great performance gain for typical real-time applications and thus we decided to exclude it.

- The instruction and/or data cache can be (partially) locked against replacement. We assume this feature is only used in certain configurations and is fixed during runtime. This adds only a slight amount of complexity and is a useful feature, so it is included in the model for the pipeline.

- The data cache must be configured for write-through mode. Write-back cannot be analyzed precisely due to the bad worst-case behavior of the PPC 755 cache.

- Effects on memory protection registers (TLB, etc) are not modeled. Virtual memory was too difficult to model for this version of the analyzer.

- Atomic update instructions for multiprocessor synchronization, **lswarx** and **stwcx** are not modeled. They can be modeled in principle but are rather useless in a real-time system with only one bus master.

- Multiple load/stores with dynamic load/store counts are not modeled, i.e. the instructions **lswx** and **stswx** must not occur in any program. These instructions would lead to a huge loss in precision of the analysis because the number of memory accesses is not known and must be approximated.

- Memory access times are assumed to be constant for each memory area. This simplifies the modeling of the chip set unit. Another version includes a more elaborated chip set unit with a controller that supports SDRAM and PCI.

2.6.6 Timing Anomalies with the PPC 755

The following two examples are taken from [Sch02]. The first example shows that an empty pipeline at the beginning of the execution of an instruction sequence may lead to a *longer* execution time of the whole sequence, than a partially filled one.

The second example shows that effects of timing anomalies are unbounded by fixed hardware constants, but can only be bounded by program length and iteration counts of loops, which makes their approximation non-local.

Empty Pipeline may be worst-case scenario

One assumption about pipelines is that a program starting with no other instructions occupying pipeline stages before it will run faster or at least as fast as were the case if the pipeline was not empty. This assumption is based on the argumentation that occupied stages ahead in the pipeline may lead to hazards for the program and thus slow down the program execution. As the MPC 755 schedules instructions dynamically, this assumption is no longer true. In fact, occupied stages ahead in the pipeline can *prevent* scheduling decisions that lead to hazards in the *program itself*.

The example program in Figure 2.14 exhibits this behavior. The program whose timing is delayed starts at index 1, the unit an instruction may be scheduled to is given in the third column. All multiplication instructions must be scheduled to IU1, the other integer instructions may be scheduled to either IU1 or IU2, whatever unit is free (IU2 takes precedence if both units are free). The operands of the multiplication instructions must be such that the multiply takes 5 cycles (upper 8 bits not zero). The RAW dependency between instructions i_{-2} and i_{-1} delays execution of the latter until the former has finished. The RAW dependency between instructions i_{-1} and i_1 also forces the latter to wait until instruction i_{-1} has finished execution.

If the instruction sequence is executed from index i_{-3} on, then instruction i_{-1} is scheduled to the IU2 unit, while i_1 is scheduled to IU1 (IU2 is not free because

Index	Instruction		Unit
i_{-3}	addi	r15,r15,4	IU1/IU2
i_{-2}	lwz	r20,0(r31)	LSU
i_{-1}	addi	r16,r20,4	IU1/IU2
i_0	fadd	f31,f31,f30	FPU
i_1	addi	r17,r16,4	IU1/IU2
i_2	lwz	r18,0(r17)	LSU
i_3	addi	r22,r18,4	IU1/IU2
i_4	addi	r21,r19,4	IU1/IU2
i_5	mullw	r25,r23,r24	IU1
i_6	lwz	r26,0(r22)	LSU
i_7	lwz	r27,0(r25)	LSU
i_8	addi	r8,r26,4	IU1/IU2
i_9	addi	r28,r25,4	IU1/IU2
i_{10}	addi	r5,r27,1	IU1/IU2
i_{11}	mullw	r4,r14,r30	IU1
i_{12}	mullw	r3,r6,r30	IU1

Figure 2.14: Timing anomaly MPC 755 I: Empty pipeline

of the RAW dependency). Following the other dependencies, instruction i_{10} is scheduled to IU2, while instruction i_{11} is scheduled to IU1 and begins execution even before instruction i_{10}. The whole sequence finishes after **19 cycles**.

If the sequence starts with instruction i_1 and an empty pipeline, then i_1 will be scheduled to IU2 (which is not occupied). Following the dependencies, instruction i_{10} must be scheduled to IU1, as IU2 is occupied by instruction i_9. Thus, instruction i_{11} must be scheduled to the reservation station of IU1, as it must execute on IU1 and the unit itself is occupied by instruction i_{10}. In this scenario, execution of instruction i_{11} begins *after* instruction i_{10} has finished and the (shorter) sequence takes **22 cycles**.

The details of the instruction scheduling for these sequences can be found in [Sch02, A.1].

Unbounded Timing Anomaly

The timing behavior of the MPC 755 is sensible to small perturbations. Given the program in Figure 2.15, when the pipeline is empty at the beginning, instruction i_2 is scheduled to IU2 and i_3 to IU1. i_2 cannot begin execution until i_1 has finished due to a RAW dependency.

Index	Instruction		Unit
i_1	lwz	r20,0(r2)	LSU
i_2	addi	r21,r20,4	IU1/2
i_3	mullw	r19,r14,r29	IU1
i_4	lwz	r23,0(r20)	LSU
i_5	addi	r24,r23,4	IU1/2
i_6	addi	r25,r14,4	IU1/2
i_7	lwz	r26,0(r19)	LSU
i_8	mullw	r27,r14,r29	IU1
i_9	lwz	r28,0(r26)	LSU
i_{10}	addi	r22,r28,0	IU1/2

Figure 2.15: Timing anomaly MPC 755 II: Unbounded effect

The whole sequence takes **10 cycles** to execute once, **19** to execute twice and **10**+$9(n-1)$ to execute n times.

If the pipeline is not empty, but one instruction occupies IU2 for 1 cycle initially, then instruction i_2 is scheduled to IU1 and instruction i_3, which must execute in IU1, must wait for the end of instruction i_2. Looking at several executions of this scenario, it takes $12n$ **cycles** to execute this sequence n times. If this sequence forms the body of a loop and is executed 100 times, then the difference in execution times is **299 cycles**, although the initial delay was just one cycle due to the added instruction.

Both scenarios presented here have been verified by measurements on a SandPoint 3 evaluation system from Motorola.

Chapter 3

Semantics and Analyses

Correctness of a hard real-time system is a crucial property and must be guaranteed. This implies that also the methods used to *verify* this correctness must meet stringent correctness and reliability goals. Therefore, every methodology used in the validation of the requirements of a real-time system must have a sound and clear basis. The way chosen here to achieve this is to define a clear (mathematical) semantics for the system to be analyzed and to prove that the analyses deployed to obtain information about the system are sound w.r.t. this semantics: if the analysis determines that a given property is true for the system then it must be true for all concrete executions of the system. In this chapter, we will present the general framework for defining semantics for a real-time system consisting of a processor and associated peripherals, together with a general framework that guarantees the correctness of analyses that fulfill certain conditions w.r.t. the semantics. The presentation is an adapted recapitulation of the foundations of program semantics and abstract interpretation.

The instantiated semantics definition for the pipeline analysis is presented in Chapter 4, the analysis based on it is defined and proven correct in Chapter 5. Another analysis that is used in the framework of our WCET tool, the value analysis, is described in Section 6.2.

It should be noted that the definition of the semantics itself must be looked at under correctness considerations: in the context in which the analyses presented in this work have been developed, authoritative information about the underlying hardware was not available. The semantics itself had to be extracted from public documents about the processors and the peripherals, combined with experimental verification and questioning of the design teams of the hardware. Thus, the analyses are only correct if this modeling of the semantics is correct w.r.t. the real hardware. This problem of validating the information the semantics is based on as well as that of validating the implementation of the analyses is discussed in more detail in Chapter 7. Possible ways to obtain a semantics (semi-)automatically from authoritative descriptions (e. g. VHDL models) are the topic of ongoing research and are discussed in Section 9.3.

3.1 Semantics

When defining the semantics of the program execution in a real-time system, we may take one of two views:

- With a *system-centric* semantics we describe how the system evolves as a whole. In our case, this evolution would be in steps of one processor cycle. Such a semantics would define a start state and an end state and then only describe the evolution of a state, beginning with the start state until the end state is reached.

- With a *instruction-centric* semantics we describe what the effects of executing this very instruction are on the state of the system.

The second view is equivalent to the first one: by composing the effects of the instructions along an execution path of the program, we obtain the whole evolution of the system. The first view can only be used to obtain the second one if we can define a predicate on the system state that can consistently determine when an instruction has finished its execution (e. g. when it has left the processor pipeline). In the presence of modern processor features like branch folding and speculative execution this is a non-trivial issue.

The instruction-centric semantics is easily utilized to denote the semantics of a machine program w.r.t. the *visible* effects on the machine programming model. Unfortunately, the machine programming model is only concerned with such things as contents of registers and memory. Timing is not part of this model. E.g., the PowerPC user model does not say anything about additional memory reads being performed by a program, as far as these reads are from normal RAM (not memory mapped control registers). A program may, and actually does on the PPC 755, perform speculative data reads via instructions that will be canceled (after a branch misprediction) later. Clearly, this influences the timing behavior significantly.

A system-centric semantics on the other hand can capture such effects better in that it can represent the evolution of the system with sufficient timing resolution. This style of semantics has the drawback that it is more difficult to make statements about *parts* of the program, as the state trace initially only describes the execution from program start to program end. Using a system-centric semantics, i. e. a pure state trace semantics, leads to an analysis that is quite similar to the one used by Lundqvist and Stenström in [LS98] which uses approximations to execution traces to obtain upper bounds to WCETs. The drawbacks of this approach have already been discussed in Section 1.4.1.

Being able to *modularize* the analysis of a program by obtaining significant information for parts of it (say, basic blocks in a machine program) and then using other techniques to combine the results (as ILP for the computation of the worst-case path through the program) is a key point for obtaining efficient analyses. And to obtain a modular analysis one needs to define the semantics in a modular way.

Luckily, the system-centric semantics can be utilized for the definition of an instruction-centric semantics associated with the instructions of the program. This is done by carefully defining when the effects of this instruction are no longer visible in the state and then assigning sub-sequences of the state trace to each instruction. This means that the effect of one instruction is represented by a sequence of states (during which this instruction is being *executed*). Clearly, in the presence of the highly parallel execution in today's processor pipelines finding a sufficient condition for the end of an instruction's execution can be challenging.

Since an instruction-centric semantics is better suited for defining the associated analyses, and because numerous techniques to do so have already been developed in the context of compiler construction theory, we will adopt this style in our semantics definitions. We will make use of different aspects of the machine semantics in two places in our WCET tool chain:

- The so-called *value analysis* [Sic97] tries to determine the addresses of data accesses that may occur during program execution. Since the addresses of data accesses can depend on register contents, we perform an *interval analysis* on the machine registers. Interval analysis is a special case of the analysis presented in [CH78] that determines linear inequalities between program variables. Here, the machine registers are the variables and the inequalities have the simple form $a \leq r \leq b$ with r being the contents of a machine register and a and b constants. This way, the interval analysis is a generalization of *constant propagation* [CCKT86] which is a standard analysis in compilers. For this analysis, we only consider the user visible effects of the semantics, which makes it much easier to define the semantics and the analysis itself. In Section 6.2 we will recapitulate how this is achieved and how the results of this analysis are used in the pipeline analysis.

- The pipeline analysis itself determines upper bounds to execution times of basic blocks in the program. To define the analysis we first define a state trace semantics. Then we address the question how meaning can be assigned to the single instructions in the program, i. e. how an instruction-centric semantics can be derived from the trace semantics. This approach relies on the assumption that we are able to define *when* an instruction has finished its execution, i. e. when it has left the pipeline. Thus, we are only able to obtain pipeline analyses for those classes of processors for which this is possible. As we will show, one is able to do so for processors that perform branch folding, speculative execution and out-of-order execution, so this assumption should not be too restrictive in practice. Chapter 4 and 5 elaborate on this in detail.

3.1.1 Program Representation

Both forms of semantics annotation require knowledge about the instruction that will be executed *after* the current instruction. The value semantics, e. g., will have different results for branch instructions, if the branch is being taken or not. Consequently, the results for the instruction immediately following the branch in the program and the one at the target address of the branch are different. Therefore, we should assign semantics not to instructions but to *pairs* of instructions, where the second instruction represents that instruction which will be executed next. This view is the same as taken in [Cou81], where the next instruction to execute is encoded in the semantic value (state) given to the semantics functions. We choose to make explicit this dependency by defining semantics and analyses on the *control-flow graph* (CFG) of a program. This graph describes (a superset of) all possible execution paths of a program. It is made up of *nodes* which are labeled with program constructs[1]. Directed, labeled, *edges* connect nodes, if the execution can pass from the source node to the target node without passing other nodes in between. With these edges, we will associate semantics.

How this CFG can be reconstructed from the executable code of a program is described in Section 6.1. Figure 3.1 shows a fragment of a program in PowerPC machine code and the corresponding CFG.

Definition 3.1.1 (CFG): A control-flow graph is a graph $G = (V, E)$, with a set V of *vertices* (or *nodes*) and a set $E \subseteq V \times V$ of *edges*. The nodes are labeled with program constructs; we write $\mathsf{prog}(v)$ for the label of a node $v \in V$. The edges, which are written as $e = v_1 \rightarrow v_2$ ($v_1, v_2 \in V$) are labeled with *edge labels*, denoted by $\mathsf{lab}(e)$ for an edge $e \in E$.

We assume that G has a unique start node and a unique end node, which are denoted by s_G and e_G, resp. In addition, there must exist a path (see below) through G from s_G to every node v. Likewise, a path must exist from every node v to e_G. ■

Note that a CFG may also contain nodes that do not correspond directly to a program construct. E.g. to ensure that unique start and end nodes do exist,

[1]In our case, the nodes are labeled with machine instructions.

```
0x10200a0:  cmpi cr0, 0x0, r3, 10
0x10200a4:  addi r0, zero, 1
0x10200a8:  bc 0xc, cr0.gt, 0x10200b0.f
0x10200ac:  addi r0, zero, 0
0x10200b0:  or r3, r0, r0
0x10200b4:  bclr 0x14
```

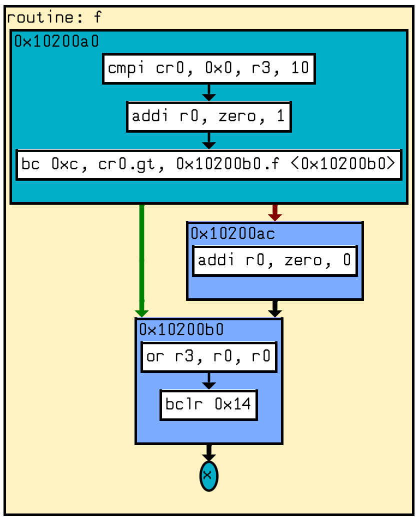


Figure 3.1: Program fragment and its CFG

we can add special nodes. These are labeled with special symbols to distinguish them from program nodes (in Figure 3.1 the node labeled x is such a node.). The **RETURN** and **CALL** nodes in Section 3.3.1 are other examples of special nodes that do not correspond to program constructs.

3.1.2 Concrete Semantics

Before we can explore the static analysis of a program and its correctness, we have to define the concrete semantics of a program. In the semantics presented here, the system is described by a state, i. e. the contents of registers, memory, etc. Execution of a program is then described by the changes to an initial state by the instructions of the program executed along a path through the CFG:

Definition 3.1.2 (State): A program execution is defined by the transformation of a state. We denote the set of all states by Σ, a single state by σ. ■

The effect of an instruction depends not only on the instruction itself but also on the next instruction to be executed. That is, rather than describing an effect of *one* instruction execution on a state, we attribute the effect to the *edge* in the CFG, connecting an instruction and (one of) its successors.

Sometimes, it is impossible to traverse a certain edge given a state. E.g. if a node in the CFG has multiple outgoing edges, corresponding to a conditional branch instruction, then a state determines that only one edge may be traversed. In such a case, the transfer functions associated to the other edges should produce an "impossible" state when applied to the input state. For this, we could introduce a special element $\perp \in \Sigma$ that denotes infeasible execution. But it will later be more convenient to capture this situation in a different manner: instead of defining transfer functions as functions from $E \to \Sigma \to \Sigma$, we take them from the domain $E \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$. The second argument to a transfer function will, however, either be the empty set, $\emptyset$, denoting infeasible execution, or a singleton set $\{\sigma\}$ containing the execution state.

Definition 3.1.3 (Transfer function): The map $\mathcal{T}_{.} : E \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ assigns a transfer function $\mathcal{T}_e : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ to every edge $e \in E$ in the CFG G. We require that $\mathcal{T}_e(\emptyset) = \emptyset$ for every e. If there are several outgoing edges $e_1, \ldots, e_n$ for a node n, then there is at most one edge e_i with $\mathcal{T}_{e_i}(x) \neq \emptyset$. ■

If $\mathcal{T}_e(\{\sigma\}) = \emptyset$ then this means that σ cannot occur as an input state on a real execution along edge e. This is just one way to capture the fact that transfer functions are partial functions. The condition that there can be at most one outgoing edge along which the execution is feasible guarantees a deterministic program execution.

With this, we can define the semantics of a program starting with a state σ as the (possibly infinite) trace of states occurring during execution:

Definition 3.1.4 (Semantics): Given a CFG G, the semantics of G is defined by a function $S : \Sigma \to \Sigma^*$, defined by

$$S(\sigma) = S'(\{\sigma\}, \mathsf{s}_G)$$

where (with $T = \{\sigma\}$)

$$S'(T, v) = \sigma . \begin{cases} \varepsilon & \text{, if } v = \mathsf{e}_G \\ S'(\mathcal{T}_e(T), v') & \text{, if } v \neq \mathsf{e}_G \wedge \exists e = v \to v' : \mathcal{T}_e(T) \neq \emptyset \\ S'(T, v) & \text{, otherwise} \end{cases}$$

■

S maps a start state σ to a (possibly infinite) state trace $\sigma_1.\sigma_2.\dots$. Execution is along the edges of the CFG, starting with s_G. At every node, we have either reached the end node e_G of the CFG and execution stops, or there is at most one outgoing edge with a transfer function that gives a non-empty result for the current state. If there is no such outgoing edge, the resulting state trace is infinite and will "loop" at the current node. If there is one valid outgoing edge, then the state trace is the current state, prolonged with the state trace from the target node with the state transformed by the transfer function.

Since the program execution is defined to be deterministic, the semantics is well defined, i. e. there is at most one edge leaving a node n, which is feasible for the continuation of the program execution. The trace is finite, if the program terminates and infinite is the program either does not terminate or gets stuck in an infeasible state (run time error).

For the argumentation of the correctness of an analysis we will make use of the *path* through the CFG that is taken during execution of the program, which is equivalent to the program trace.

Definition 3.1.5 (Path): Let $G = (V,E)$ be a CFG. A (infinite) path π in G is a sequence of edges, written $e_1.e_2.\cdots$, where for two consecutive edges e_i, e_{i+1} it holds that $\exists v_j, v_k, v_l \in V : e_i = v_j \to v_k, e_{i+1} = v_k \to v_l$. The set of all paths (in G) is denoted by Π_∞; the empty path by ε.

The set of all *finite* paths in G is denoted by Π, $\Pi \subseteq \Pi_\infty$.

The set of all paths in G, which start at node s and end in node v is written $\mathsf{P}(s,v) \subseteq \Pi$.

For every path $\pi \in \Pi_\infty$, the starting node of π is written as $\mathsf{start}(\pi)$; $\mathsf{start}(\pi) = v_1 \iff \pi = v_1 \to v_2.e_2.\cdots$.

Likewise, we write $\mathsf{end}(\pi)$ for the ending node of a finite path $\pi \in \Pi$, $\mathsf{end}(\pi) = v_n \iff \pi = e_1.\cdots.v_{n-1} \to v_n$, $n > 0$. ■

The path taken during the execution of the program is already available in the definition of the semantics: we only have to collect the edges e of the transfer functions giving the successor state to one state in the trace. The function $p_G : \Sigma \to \Pi_\infty$ gives this path:

$$p_G(\sigma) = p'_G(\{\sigma\}, \mathsf{s}_G)$$

where

$$p'_G(S,v) = \begin{cases} \varepsilon & \text{, if } v = \mathsf{e}_G \\ e.p'_G(\mathcal{T}_e(S), v') & \text{, if } v \neq \mathsf{e}_G \wedge \exists e = v \to v' : \mathcal{T}_e(S) \neq \emptyset \\ v \to v.p'_G(S,v) & \text{, otherwise} \end{cases}$$

From a given path π, we can deduce the sequence of states that occur during a walk along this path. For every finite path, the *path semantics* determines the effect on a start state of executing the program along this path:

Definition 3.1.6 (Path semantics): Let $\pi \in \Pi$ be a finite path. Define the path semantics $[\![\cdot]\!] : \Pi \to \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ as

$$[\![e_1.\cdots.e_n]\!] := \mathcal{T}_{e_n} \circ \cdots \circ \mathcal{T}_{e_1}$$

■

Naturally, we cannot give a semantics to an infinite path. We can, however, give a semantics to every finite prefix of such a path, which is sufficient for our needs. Thus, if the program G starting from σ terminates and the last state in $S(\sigma)$ is σ', then we also have $\sigma' = [\![p_G(\sigma)]\!](\{\sigma\})$.

As we want to obtain information for *parts* of the program later, we have to look not only at the state *after* execution of the program but also at the states *during* execution. Therefore, we associate with every program point, i. e. instruction, the set of states that occur during execution at that point. Since program points correspond to the nodes in our CFG, we define the set of possible states that may occur at each node of the CFG as a map from CFG nodes to the powerset of states. This can easily be done by gathering the path semantics of every prefix of the execution path that ends in this node:

Definition 3.1.7 (State map): Let σ be an initial start state for program execution and $\pi = p_G(\sigma)$, then define the map $\mathsf{states}^\sigma : V \to \mathcal{P}(\Sigma)$ as

$$\mathsf{states}^\sigma(v) := \bigcup\{[\![\pi']\!]\{\sigma\} \mid \pi' \in \mathsf{prefixes}(\pi) \cap \mathsf{P}(\mathsf{s}_G, v)\}$$

where $\mathsf{prefixes}(\pi)$ is the set of all finite prefixes of the path π. ■

states^σ gives for every start state σ the set of all states that occur at a node during execution.

If we consider only *one* execution of the program, starting from a given state, then any analysis based on this can only give information about that particular execution. Our analyses will give results that are valid for *all* executions of the program. Therefore we first define the collection of all program executions, the *collecting semantics*:

Definition 3.1.8 (Collecting semantics): We define the collecting semantics $\mathcal{A}_G^{\mathsf{co}} : \mathcal{P}(\Sigma) \to (V \to \mathcal{P}(\Sigma))$ (where V are the nodes from the CFG G) as

$$\mathcal{A}_G^{\mathsf{co}}(S) = \lambda v. \bigcup_{\sigma \in S} \mathsf{states}^\sigma(v) \tag{3.1.9}$$

■

If the set S contains all possible start states, then $\mathcal{A}_G^{\mathsf{co}}(S)$ gives all possible states at each program point that can occur during execution.

Everything that can be said about the program G is represented by this collecting semantics. Naturally, it is not computable in general; even if we restrict ourselves to terminating programs and finitely many possible start states, we cannot compute $\mathcal{A}_G^{\mathsf{co}}$ efficiently. Therefore, we have to search for approximate descriptions of the information represented by $\mathcal{A}_G^{\mathsf{co}}$; such an approximation will be called *analysis* of G. This information will necessarily be incomplete but is required to be *safe*: if an approximation (analysis) says that a given property at a program point v holds, then this property must be satisfied by every state $\sigma \in \mathcal{A}_G^{\mathsf{co}}(S)(v)$. The next section presents the framework of *abstract interpretation* which defines some general relations between $\mathcal{A}_G^{\mathsf{co}}$ and any approximation. Section 3.3 then presents an *implementation* method for analyses, the *data-flow analysis*, and shows that it satisfies the necessary correctness constraints.

3.2 Program Analysis

This section presents the theory of static program analysis in the framework of abstract interpretation, [CC77, CC79, Cou81, CC91, CC92a, CC92b, NNH99]. Due to the high importance of the correctness of analysis results in real-time systems, the presentation is quite detailed.

3.2.1 Abstract Interpretation

The framework of abstract interpretation defines relations between a concrete semantics (here the collecting semantics) and an abstract semantics. While the collecting semantics computes a set of states for each program point, thus works on the *domain* $\mathcal{P}(\Sigma)$, an abstract semantics computes values on an *abstract domain* $\hat{D}$. Abstract values $\hat{d}$ from this domain can be seen as *approximations* of the set of states computed in the collecting semantics.

We assume that there is an ordering on abstract values, that captures the *precision* of the approximation represented by the abstract values. Therefore, we first define some concepts about orderings and ordered sets that will be utilized in the sequel.

Definition 3.2.1 (Partially Ordered Set): Given a set A, a relation $\sqsubseteq_A \subseteq A \times A$ is called partial ordering iff it is reflexive, transitive and anti-symmetric. We call $(A, \sqsubseteq_A)$ a partially ordered set. ■

Definition 3.2.2 (Bounds): For a partially ordered set $(A, \sqsubseteq_A)$ and a set $Y \subseteq A$ we call an element $a \in A$ lower bound of Y, iff $\forall y \in Y : a \sqsubseteq_A y$. Likewise we call $b \in A$ upper bound of Y iff $\forall y \in Y : y \sqsubseteq_A b$. We call an element $l \in A$ greatest lower bound (or glb for short) of Y if it is a lower bound for Y and for all other lower bounds a of Y we have $a \sqsubseteq_A l$. We call an element $u \in A$ least upper bound (or lub for short) of Y if it is an upper bound for Y and for all other upper bounds b of Y we have $u \sqsubseteq_A b$. If greatest lower bounds or least upper bounds exist, they are unique and we write $\bigsqcap Y$ for the glb of Y and $\bigsqcup Y$ for the lub of Y. ■

Definition 3.2.3 (Complete Lattice): A partially ordered set $(L, \sqsubseteq_L)$ is called a complete lattice if $\bigsqcup Y$ and $\bigsqcap Y$ exist for every $Y \subseteq L$. A complete lattice has a least element $\bot \in L$ and a greatest element $\top \in L$. For these we have $\bot = \bigsqcup \emptyset$ and $\top = \bigsqcup L$. For convenience we write $a \sqcup b$ for $\bigsqcup \{a, b\}$ and $a \sqcap b$ for $\bigsqcap \{a, b\}$. $\sqcap$ is called meet operator and $\sqcup$ is called join operator. To define a complete lattice we sometimes write it as a tupel $(L, \sqsubseteq_L, \bot, \top, \bigsqcup, \bigsqcap)$. ■

We will use a partially ordered set $(A, \sqsubseteq_A)$ to capture the precision of the elements from A. If we have $a, b \in A$ with $a \sqsubseteq_A b$ then we say that a is *more precise* than b; or that b is *more abstract* than a.

If $(A, \sqsubseteq)$ is even a complete lattice, then we can find for any set $Y \subseteq A$ of approximations the *most precise one*, a, which is $a = \bigsqcap Y$.

Example 3.2.4 (Collecting Semantics): The domain used in the collecting semantics, $\mathcal{P}(\Sigma)$ together with the subset ordering $\subseteq$ is a complete lattice $(\mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \bigcup, \bigcap)$.

Here, if $S \subseteq S'$ for subsets S, S' of Σ, then S' is more abstract because it contains more states than S and we usually cannot make more precise statements that hold for all states in S' than we can for the states in S.

□

Example 3.2.5 (Intervals): The set $\mathbb{N}^\top = \mathbb{N} \cup \{\top\}$ together with the ordering $\sqsubseteq_\top$ defined by

$$\begin{array}{l} a \sqsubseteq_\top \top \\ a \sqsubseteq_\top b \text{, if } a \neq \top \wedge b \neq \top \wedge a \leq b \end{array}$$

is a partially ordered set.

The set $I = \mathbb{N}^\top \times \mathbb{N}^\top$ of *intervals* together with the ordering $\sqsubseteq_I$ defined by $(a, b) \sqsubseteq_I (c, d)$ iff $c \sqsubseteq_\top a \wedge b \sqsubseteq_\top d$ is a complete lattice with

$$\begin{array}{l} \bot_I = (\top, 0) \\ \top_I = (0, \top) \\ \bigsqcup Y = (\min\{a \mid (a,b) \in Y\}, \max\{b \mid (a,b) \in Y\}) \\ \bigsqcap Y = (\max\{a \mid (a,b) \in Y\}, \min\{b \mid (a,b) \in Y\}) \end{array}$$

This complete lattice can be used to approximate a set of natural numbers. The set of numbers represented exactly by an interval (a,b) is $\{n \mid a \leq n \leq b\}$. Likewise, for any set of numbers N, we can give intervals that represent *at least* these numbers. Since I is a complete lattice we can even give a most precise (i. e. smallest w.r.t. $\sqsubseteq_I$) interval that contains all numbers from N: $(\min N, \max N)$.

□

Analysis will later give approximations, i. e. values from abstract domains instead of a set of states as does the collecting semantics. These values must be related to the results of the collecting semantics. To formalize this relation, we can use a *concretization* function $\gamma : \hat{D} \to \mathcal{P}(\Sigma)$that maps abstract values to the set of states they approximate (or represent). We fix some notations for functions working on partially ordered sets:

Definition 3.2.6 (Functions): Let $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ be partially ordered sets. A function $f : A \to B$ is called monotone iff $a \sqsubseteq_A a' \Rightarrow f(a) \sqsubseteq_B f(a')$. It is called surjective, if $\forall b \in B : \exists a \in A : b = f(a)$, i. e. every element in B can be reached from A via f. If $(A, \sqsubseteq_A)$ and $(B, \sqsubseteq_B)$ are complete lattices, we call f distributive (or completely additive) iff for all subsets $Y \subseteq A$: $f(\bigsqcup_A Y) = \bigsqcup_B \{f(a) \mid a \in Y\}$. Likewise we call f completely multiplicative iff $f(\bigsqcap_A Y) = \bigsqcap_B \{f(a) \mid a \in Y\}$. ■

Clearly, γ must be monotone: the concretization of more abstract values must also be more abstract in the collecting semantics. And for every concrete value there must be an abstract one whose concretization approximates the concrete value: $\forall S \in \mathcal{P}(\Sigma) : \exists \hat{d} \in \hat{D} : S \subseteq \gamma(\hat{d})$

An analysis is now just an abstract semantics for a program G, working on a domain $\hat{D}$ of abstract values. The analysis gives for each program point an abstract value:

Definition 3.2.7 (Analysis): Given a partially ordered set $(\hat{D}, \sqsubseteq_{\hat{D}})$ we call any monotone map $\mathcal{A}_G : \hat{D} \to (V \to \hat{D})$ an analysis on the program $G = (V, E)$. ■

Thus, $\mathcal{A}_G^{\mathsf{co}}$ can be viewed as an analysis (the most precise one), too. The notion of precision via the partially orderings is also visible in the definition of the collecting semantics: it is monotone in its first argument on the domain $\mathcal{P}(\Sigma)$ with the ordering $\subseteq$, if we define the complete lattice $(V \to A, \sqsubseteq_\to)$ by

Definition 3.2.8 (Function Lattice): Let V be a set, $(A, \sqsubseteq_A)$ a complete lattice, then the functional lattice $(V \to A, \sqsubseteq_\to)$ is a complete lattice with $f \sqsubseteq_\to$

$g \iff \forall v \in V : f(v) \sqsubseteq_A g(v)$. $\top_\rightarrow = \lambda v.\top_A$ and $\bot_\rightarrow = \lambda v.\bot_A$. The join and meet operators are defined by $\bigsqcup_\rightarrow F = \lambda v.\bigsqcup_A\{f(v) \mid f \in F\}$ and $\bigsqcap_\rightarrow F = \lambda v.\bigsqcap_A\{f(v) \mid f \in F\}$, where $F \subseteq V \rightarrow A$. If $(A, \sqsubseteq_A)$ is not a complete lattice but a partially ordered set, then so is $(V \rightarrow A, \sqsubseteq_\rightarrow)$. ■

The monotonicity of $\mathcal{A}_G^{\mathsf{co}}$ means that we obtain less precise concrete results if we have less precise inputs: we cannot gain more precise information that is true for all states at a program point from a set S' of inputs, than we can gain for a set $S \subseteq S'$ of inputs. Thus, $\mathcal{A}_G^{\mathsf{co}}$ is in itself an analysis with $\hat{D} = \mathcal{P}(\Sigma)$.

With the monotone concretization function γ we can relate the results of an analysis with those of the collecting semantics. A correct analysis will deliver approximations that represent *at least* all concrete states that can occur during execution, if the input approximation represents all possible inputs to the concrete program execution:

Definition 3.2.9 (Correct Analysis): Let $\mathcal{A}_G : \hat{D} \rightarrow (V \rightarrow \hat{D})$ be an analysis of G on the partially ordered set $\hat{D}$ and $\gamma : \hat{D} \rightarrow \mathcal{P}(\Sigma)$ be a concretization. We say that $\mathcal{A}_G$ is a correct analysis (w.r.t. $\mathcal{A}_G^{\mathsf{co}}$ and γ) of G iff

$$\forall S \subseteq \Sigma, \hat{d} \in \hat{D} : S \subseteq \gamma(\hat{d}) \Rightarrow \mathcal{A}_G^{\mathsf{co}}(S) \sqsubseteq_\rightarrow \gamma \circ \mathcal{A}_G(\hat{d}) \qquad (3.2.10)$$

■

In fact, we can generalize this correctness approach by defining correctness not w.r.t. $\mathcal{A}_G^{\mathsf{co}}$ but a different analysis $\mathcal{B}_G : \hat{E} \rightarrow (V \rightarrow \hat{E})$ working on a domain $\hat{E}$:

Definition 3.2.11 (General Correct Analysis): Let $\mathcal{B}_G : \hat{E} \rightarrow (V \rightarrow \hat{E})$ and $\mathcal{A}_G : \hat{D} \rightarrow (V \rightarrow \hat{D})$ be analyses of G on the partially ordered sets $\hat{E}$ and $\hat{D}$ respectively. And let $\gamma : \hat{E} \rightarrow \hat{D}$ be a concretization function. We call $\mathcal{B}_G$ a correct analysis of G w.r.t. $\mathcal{A}_G$ iff

$$\forall \hat{d} \in \hat{D}, \hat{e} \in \hat{E} : \hat{d} \sqsubseteq_{\hat{D}} \gamma(\hat{e}) \Rightarrow \mathcal{A}_G(\hat{d}) \sqsubseteq_\rightarrow \gamma \circ \mathcal{B}_G(\hat{e}) \qquad (3.2.12)$$

■

From now on we will omit the program G from the denotation of analyses, so we write $\mathcal{A}$ instead of $\mathcal{A}_G$.

One advantage of abstract interpretation is now that we can *combine* analyses and still have a correct analysis. Suppose we have three domains $\hat{D}$, $\hat{E}$ and $\hat{F}$ and analyses $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, working on $\hat{D}$, $\hat{E}$ and $\hat{F}$ resp. Additionally, $\mathcal{B}$ is correct w.r.t. $\mathcal{A}$ and $\mathcal{C}$ is correct w.r.t. $\mathcal{B}$. Let the concretizations be $\gamma_1 : \hat{E} \rightarrow \hat{D}$ and $\gamma_2 : \hat{F} \rightarrow \hat{E}$. Then $\mathcal{C}$ is also correct w.r.t. $\mathcal{A}$ with the concretization $\gamma = \gamma_1 \circ \gamma_2$:

$$\forall \hat{d} \in \hat{D}, \hat{e} \in \hat{E} : \hat{d} \sqsubseteq_{\hat{D}} \gamma_1(\hat{e}) \Rightarrow \mathcal{A}(\hat{d}) \sqsubseteq_\rightarrow \gamma_1 \circ \mathcal{B}(\hat{e}) \wedge$$
$$\forall \hat{e} \in \hat{E}, \hat{f} \in \hat{F} : \hat{e} \sqsubseteq_{\hat{E}} \gamma_2(\hat{f}) \Rightarrow \mathcal{B}(\hat{e}) \sqsubseteq_\rightarrow \gamma_2 \circ \mathcal{C}(\hat{f})$$

Because γ_1 is monotone the conjunction of the two assumptions implies $\forall \hat{d}, \hat{f}$: $\hat{d} \sqsubseteq_{\hat{D}} \gamma_1(\gamma_2(\hat{f}))$ and the conclusion yields $\mathcal{A}(\hat{d}) \sqsubseteq_{\rightarrow} \gamma_1 \circ \gamma_2 \circ (\mathcal{C}(\hat{f}))$, i.e. $\mathcal{C}$ is correct w.r.t. $\mathcal{A}$ with $\gamma = \gamma_1 \circ \gamma_2$, as the combination of two monotone functions is again monotone.

The ability to obtain a correct analysis from the combination of two correct analyses can be very convenient for the design of the domains, analyses and concretizations. It may be easier to define a domain and analysis "in between" the collecting semantics and the final domain/analysis and to prove this analysis correct than to prove the correctness of the final analysis w.r.t. $\mathcal{A}^{\mathsf{CO}}$. We will now introduce such an intermediate analysis that makes it easier to prove the correctness of our data-flow analyses later. We will call this analysis the *Coarse Analysis*. It uses the same domain as the collecting semantics but does not compute the set of states at a node by considering all *feasible* paths but it gathers the results along *all* paths from the start node to a node:

Definition 3.2.13 (Coarse Analysis): For every transfer function $\mathcal{T}_e$ we define the transfer function $\mathcal{T}'_e : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ by $\mathcal{T}'_e(S) = \bigcup\{\mathcal{T}_e(\{\sigma\}) \mid \sigma \in S\}$.

We define the extended path semantics $[\![\cdot]\!]' : \Pi \rightarrow \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ by

$$[\![e_1.\cdots.e_n]\!]' = \mathcal{T}'_{e_n} \circ \cdots \circ \mathcal{T}'_{e_1}$$

The coarse analysis $\mathcal{A}^{\mathsf{C}} : \mathcal{P}(\Sigma) \rightarrow (V \rightarrow \mathcal{P}(\Sigma))$ is given by

$$\mathcal{A}^{\mathsf{C}}(S) = \lambda v. \bigcup_{\pi \in \mathsf{P}(\mathsf{s}_{G}, v)} [\![\pi]\!]'(S) \tag{3.2.14}$$

■

Note that $\mathcal{A}^{\mathsf{C}}_G$ is monotone in its first argument because the extended path semantics is monotone as the composition of the monotone functions $\mathcal{T}'_e$.

This analysis is less precise than the collecting semantics, but still correct, we just choose $\gamma : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ as $\gamma = \lambda S.S$, the identity function:

Lemma 3.2.15 (Correctness of $\mathcal{A}^{\mathsf{C}}$): $\mathcal{A}^{\mathsf{C}}$ is correct w.r.t. $\mathcal{A}^{\mathsf{CO}}$ with $\gamma = \lambda S.S$.

Proof:
We just have to show that

$$\mathcal{A}^{\mathsf{CO}}(S)(v) \subseteq \mathcal{A}^{\mathsf{C}}(S)(v)$$

holds for an arbitrary $S \in \mathcal{P}(\Sigma), v \in V$.

This can be written as

$$\bigcup_{\substack{\sigma\in S\\ \pi=p_G(\sigma)}} \{[\pi'](\sigma\} \mid \pi' \in \mathsf{prefixes}(\pi)\cap \mathsf{P}(\mathsf{s}_G, v)\} \subseteq \bigcup_{\pi'\in \mathsf{P}(\mathsf{s}_{G,v})} [\![\pi']\!]'(S)$$

Obviously, it is always correct for a path $\pi' \in \mathsf{prefixes}(\pi)\cap \mathsf{P}(\mathsf{s}_G, v)$ that $\pi' \in \mathsf{P}(\mathsf{s}_G, v)$ holds.

Thus, it is sufficient to show $[\![\pi']\!]\{\sigma\} \subseteq [\![\pi']\!]'(S)$ for $\sigma \in S, \pi = p_G(\sigma)$ and $\pi' \in \mathsf{P}(\mathsf{s}_G, v)$.

But this is the case, since a short look at the definitions shows that $[\![\pi']\!]\{\sigma\} = [\![\pi']\!]'(\{\sigma\})$. And for $[\![\pi']\!]'$ it is true that $S \subseteq S' \Rightarrow [\![\pi']\!]'(S) \subseteq [\![\pi']\!]'(S')$.

□

The coarse analysis is useful, because it is in general not possible to effectively compute the set of *feasible* paths to a program point. The coarse analysis simply ignores the feasibility constraint and determines the state map using *all* paths to the program point. All feasible executions are included by this approach, but erroneous states are added at each program point, thus this analysis is less precise than the collecting semantics. Since analyses can normally only compute information about all paths, not just the feasible ones, the coarse analysis helps to prove these analyses correct.

Galois Connections

In Definition 3.2.11, the correctness of an analysis $\mathcal{B}_G : \hat{E} \to (V \to \hat{E})$ is defined w.r.t. an analysis $\mathcal{A}_G : \hat{D} \to (V \to \hat{D})$. In essence $\mathcal{B}_G$ is correct, if its results starting from an $\hat{e} \in \hat{E}$ are correct approximations to the results of $\mathcal{A}_G$ starting from a $\hat{d} \in \hat{D}$, and $\hat{e}$ is an approximation of $\hat{d}$. The question remains, that given a $\hat{d}$ for $\mathcal{A}_G$, *which* $\hat{e}$ to choose as the start of $\mathcal{B}_G$. Naturally, it should be as precise as possible to give the most precise analysis results for $\mathcal{B}_G$. Abstract interpretation gives us a means by which we can determine the most precise starting value for an analysis by defining a counter part to γ, namely $\alpha : \hat{D} \to \hat{E}$, the *abstraction* function. α and γ convert between the domains of the two analyses under consideration and must form a *Galois Connection* to ensure correctness:

Definition 3.2.16 (Galois connection): Let $\hat{D}$ and $\hat{E}$ be complete lattices. Let $\alpha : \hat{D} \to \hat{E}$ and $\gamma : \hat{E} \to \hat{D}$ be monotone maps. $(\hat{D}, \alpha, \gamma, \hat{E})$ is called Galois connection iff

$$\forall \hat{d} \in \hat{D} : \hat{d} \sqsubseteq_{\hat{D}} \gamma(\alpha(\hat{d})) \tag{3.2.17}$$

and

$$\forall \hat{e} \in \hat{E} : \alpha(\gamma(\hat{e})) \sqsubseteq_{\hat{E}} \hat{e} \tag{3.2.18}$$

We call α **abstraction** and γ **concretization.** ■

A Galois connection guarantees that by abstracting and going back by concretization results in an equally or less precise element, cf. Figure 3.2, where the ordering is represented by the vertical position of the elements.

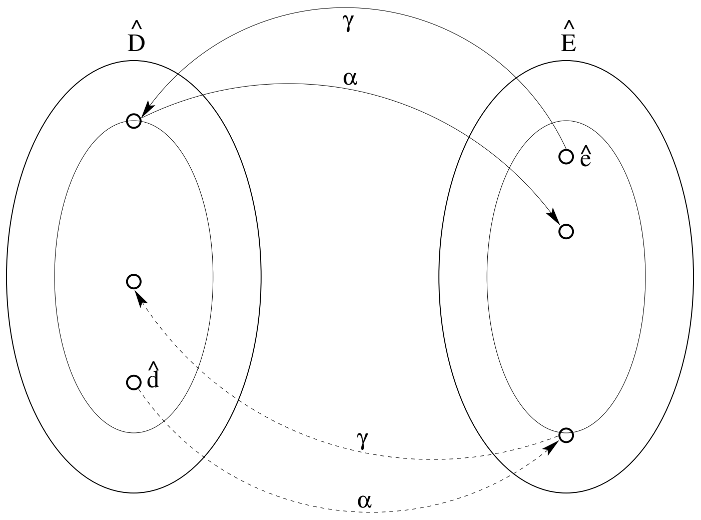


Figure 3.2: Galois connection between two domains

A trivial example is the Galois Connection connecting the domains of the collecting semantics and the coarse analysis:

Example 3.2.19 (GC Coarse Analysis): The tupel $(\hat{D}, \lambda\hat{d}.\hat{d}, \lambda\hat{d}.\hat{d})$ is a Galois Connection. Thus, the most precise input value for the coarse analysis is $\alpha(S) = S$, if S is the set of all inputs.

□

A less trivial example is the Galois Connection between $\mathcal{P}(\mathbb{N}^\top)$ and the interval domain $\mathbb{N}^\top \times \mathbb{N}^\top$:

Example 3.2.20 (Interval Galois Connection): The two complete lattices defined by $(\mathcal{P}(\mathbb{N}^\top), \subseteq)$ and $(\mathbb{N}^\top \times \mathbb{N}^\top, \sqsubseteq_I)$ together with the functions

$$\begin{array}{rcl} \gamma_I(a,b) & = & \{n \mid a \sqsubseteq_\top n \sqsubseteq_\top b\} \\ \alpha_I(N) & = & (\min N, \max N) \end{array}$$

form a Galois Connection.

$\square$

Apart from Definition (3.2.16) there is an equivalent way of determining if a pair of functions α, γ form a Galois Connection:

Lemma 3.2.21 (Adjunction): Let $\alpha : \hat{D} \to \hat{E}$ and $\gamma : \hat{E} \to \hat{D}$ be maps on the complete lattices $\hat{D}$ and $\hat{E}$. Then the following statements are equivalent:

1. $\forall \hat{d} \in \hat{D}, \hat{e} \in \hat{E} : \alpha(\hat{d}) \sqsubseteq_{\hat{E}} \hat{e} \iff \hat{d} \sqsubseteq_{\hat{D}} \gamma(\hat{e})$
2. $(\hat{D}, \alpha, \gamma, \hat{E})$ is a Galois connection.

Proof: See [NNH99].

$\square$

This equivalence between α, γ forming an adjunction and them being a Galois Connection will be utilized in a proof later. Also, we will need the fact that α is *distributive* in the correctness proof for the data-flow analysis:

Lemma 3.2.22 (Distributivity): Let $(\hat{D}, \alpha, \gamma, \hat{E})$ be a Galois connection. Then α is distributive, i. e.

$$\forall Y \subseteq \hat{D} : \alpha(\bigsqcup Y) = \bigsqcup \{\alpha(\hat{d}) \mid \hat{d} \in Y\}$$

Proof:
From Lemma 3.2.21 we have that $\forall \hat{d}, \hat{e} : \alpha(\hat{d}) \sqsubseteq_{\hat{E}} \hat{e} \iff \hat{d} \sqsubseteq_{\hat{D}} \gamma(\hat{e})$.

$$\begin{array}{rl} & \alpha(\bigsqcup Y) \sqsubseteq_{\hat{E}} \hat{e} \\ \iff & \bigsqcup Y \sqsubseteq_{\hat{D}} \gamma(\hat{e}) \\ \iff & \forall \hat{d} \in Y : \hat{d} \sqsubseteq_{\hat{D}} \gamma(\hat{e}) \\ \iff & \forall \hat{d} \in Y : \alpha(\hat{d}) \sqsubseteq_{\hat{E}} \hat{e} \\ \iff & \bigsqcup \{\alpha(\hat{d}) \mid \hat{d} \in Y\} \sqsubseteq_{\hat{E}} \hat{e} \end{array}$$

Since this equivalence holds for all $\hat{e}$ it is especially true for $\hat{e} = \alpha(\bigsqcup Y)$ and $\hat{e} = \bigsqcup \{\alpha(\hat{d}) \mid \hat{d} \in Y\}$, which, together with the asymmetry of $\sqsubseteq_{\hat{E}}$, proves the claim.

$\square$

Interestingly, γ already uniquely determines the α of the Galois Connection if γ is a complete meet morphism, i. e. is completely multiplicative:

Lemma 3.2.23 (Galois Connection Facts): If $(\hat{D}, \alpha, \gamma, \hat{E})$ is a Galois connection then

- γ is completely multiplicative, i. e.

$$\bigsqcap_{\hat{D}}\{\gamma(\hat{e}) \mid \hat{e} \in Y\} = \gamma(\bigsqcap_{\hat{E}} Y)$$

If γ is completely multiplicative, then

- γ uniquely defines α by

$$\alpha(\hat{d}) = \bigsqcap_{\hat{E}}\{\hat{e} \mid \hat{d} \sqsubseteq_{\hat{D}} \gamma(\hat{e})\}$$

Proof:
See [NNH99] lemma 4.22.

□

Also, one can show that a completely additive function α uniquely determines a γ (and thus a Galois connection) as the least upper bound of all values from $\hat{D}$ whose abstraction is more precise than the argument to γ. An easy way to determine if there exists a Galois connection for a given γ is to check if γ is completely multiplicative.

Having a Galois Connection for two or more domains, one can construct Galois Connections for various combinations of the domains, e. g. cross product, function spaces, etc. We will later need the following Galois connection for the state maps based on a Galois connection for the domains of two analyses:

Lemma 3.2.24 (Node map): Let $(\hat{D}, \alpha, \gamma, \hat{E})$ be a Galois connection. Then we have that $(V \to \hat{D}, \alpha_{\to}, \gamma_{\to}, V \to \hat{E})$ is also a Galois connection, with

$$\begin{aligned} \alpha_{\to}(f) &:= \alpha \circ f \\ \gamma_{\to}(g) &:= \gamma \circ g \end{aligned}$$

Proof:
Obviously, $\alpha_{\to}$ and $\gamma_{\to}$ are monotone, since α and γ are monotone. We have for all $v \in V, f \in V \to \hat{D}$:

$$f(v) \sqsubseteq_{\hat{D}} \gamma(\alpha(f(v))) = \gamma(\alpha_{\to}(f)(v)) = \gamma_{\to}(\alpha_{\to}(f))(v)$$

Which implies $f \sqsubseteq_{\to} \gamma_{\to}(\alpha_{\to}(f))$.

For all $v \in V, g \in V \to \hat{E}$ we have:

$$\alpha(\gamma(g(v))) = \alpha(\gamma_{\to}(g)(v)) = \alpha_{\to}(\gamma_{\to}(g))(v) \sqsubseteq_{\hat{E}} g(v)$$

From this we obtain $\alpha_{\to}(\gamma_{\to}(g)) \sqsubseteq_{\to} g$, as desired.

□

Having a Galois connection for two domains not only allows to determine the best input value for an analysis on the target domain, but even allows to *define the best analysis* possible with these domains and concretization/abstraction. This result is obtained by giving a different correctness condition for an analysis with the following theorem:

Theorem 3.2.25 (Correctness): If $\mathcal{A} : \hat{D} \to (V \to \hat{D})$ is an analysis and furthermore $(\hat{D}, \alpha, \gamma, \hat{E})$ is a Galois connection, then every analysis $\mathcal{B} : \hat{E} \to (V \to \hat{E})$ is correct w.r.t. $\mathcal{A}$ if it fulfills

$$\alpha_{\to} \circ \mathcal{A} \circ \gamma \sqsubseteq_{\to} \mathcal{B} \tag{3.2.26}$$

where $(V \to \hat{D}, \alpha_{\to}, \gamma_{\to}, V \to \hat{E})$ is as in Lemma 3.2.24.

Proof:
See [NNH99].

□

Correctness by this theorem implies correctness as in Definition (3.2.11):

$$\begin{array}{ll}
 & \alpha_{\to} \circ \mathcal{A} \circ \gamma \sqsubseteq_{\to} \mathcal{B} \\
\Rightarrow & \forall \hat{e} \in \hat{E} : \alpha_{\to}(\mathcal{A}(\gamma(\hat{e}))) \sqsubseteq_{\to} \mathcal{B}(\hat{e}) \\
\Rightarrow & \forall v \in V : \alpha(\mathcal{A}(\gamma(\hat{e}))(v)) \sqsubseteq_{\hat{E}} \mathcal{B}(\hat{e})(v) \\
\Rightarrow & \mathcal{A}(\gamma(\hat{e}))(v) \sqsubseteq_{\hat{D}} \gamma(\mathcal{B}(\hat{e})(v)) \\
\Rightarrow & \forall \hat{d} \sqsubseteq \gamma(\hat{e}) : \mathcal{A}(\hat{d})(v) \sqsubseteq_{\hat{D}} \gamma(\mathcal{B}(\hat{e})(v)) \\
\Rightarrow & \mathcal{A}(\hat{d}) \sqsubseteq_{\hat{D}} \gamma \circ \mathcal{B}(\hat{e})
\end{array}$$

by using the monotonicity of γ in step three.

Rather than verifying that a given analysis $\mathcal{B}$ satisfies condition (3.2.26), we can *define* our analysis $\mathcal{B}$ as $\mathcal{B} = \alpha_{\to} \circ \mathcal{A} \circ \gamma$. This way we not only obtain a correct analysis, but also the *most precise* one possible under the given Galois Connection. Herein lies the great advantage of using Galois connections for the definition of analyses.

Unfortunately, it is not always possible to define a Galois connection for a given γ, e. g. by using Lemma (3.2.23). That means, γ may not always be completely multiplicative. One example, which is derived from a similar argument in [CC92a] is the approximation of points in the plane by a rotated square:

Example 3.2.27 (Rotated Square): Let the points of the plane $\mathbb{R} \times \mathbb{R}$ be approximated by a square of side length a whose center is at a point x, y and which is rotated around this point by an angle $0 \leq \phi < \pi/2$ (We have to add special symbols $\perp$ and $\top$ to $\mathbb{R}$ to make them complete lattices, similar to $\mathbb{N}^{\top}$, but this is not important for this example). The concretization

$\gamma : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathcal{P}(\mathbb{R} \times \mathbb{R})$ describes the set of points enclosed by the rotated square (a, x, y, ϕ). If there was a Galois connection then there had to exist an α that is defined by (cf. Lemma (3.2.23))

$$\alpha(P) = \bigsqcap \{(a, x, y, \phi) \mid P \subseteq \gamma(a, x, y, \phi)\}$$

That is, there must be a *best* approximation to a set P of points. Looking at $P = \{(c, d) \mid |c^2 + d^2| \leq 1\}$, i. e. a circle of radius 1 around the origin, we can see that every square centered at the origin with $a = 2$ and an *arbitrary* rotation angle ϕ encloses this circle. However, different values of ϕ lead to squares enclosing different sets of points. Thus, we cannot define one square to be the *best* square approximating the circle.

□

Another example, where it is not possible to define a best abstraction is the cache analysis for the PowerPC 755 cache. This means that we will only be able to define a Galois connection for the value analysis used in our WCET toolchain but not for the PowerPC pipeline analysis, as it contains the PowerPC cache analysis.

The next section presents the type of analyses implemented in our toolchain, the data flow analyses.

3.3 Data-Flow Analysis

Data-flow analysis (DFA) is a special case of a program analysis where abstract values are propagated along the edges of the CFG and are transformed by abstract versions of the transfer functions until a fixed point of the node map is reached.

If the abstract transfer functions fulfill certain conditions w.r.t. the coarse semantics (or another analysis against which correctness should be shown) then the computed fixed point is a correct analysis.

In the following we will start by presenting the *Meet Over all Paths* (MOP) formulation of DFA, as an analysis. After we have shown the correctness of this analysis, we will repair the principal drawback of the MOP analyses: their potential uncomputability. We do this by providing a safe approximation to the analyses, the so called *Minimum Fixed Point* (MFP) analyses. These analyses are computable, if the underlying lattice fulfills the finite ascending chains condition and if the abstract transfer functions are monotone.

This section finishes with some explanations on increasing the precision of the analysis results by doing *inter-procedural* DFA. We will utilize this to increase the precision of critical loops in the program, thus giving tighter bounds for the WCET.

Definition 3.3.1 (MOP analysis): Let $\hat{D}$ be the complete lattice of the DFA. We call monotone functions $\hat{\mathcal{T}}_{\cdot} : E \to \hat{D} \to \hat{D}$ abstract transfer functions.

Let $\widehat{[\cdot]} : \Pi \to \hat{D} \to \hat{D}$ be defined by

$$\widehat{[e_1.\cdots.e_n]} := \hat{\mathcal{T}}_{e_n} \circ \cdots \circ \hat{\mathcal{T}}_{e_1}$$

The MOP-analysis $\mathsf{MOP} : \hat{D} \to (V \to \hat{D})$ is defined as

$$\mathsf{MOP} = \lambda \hat{d}.\lambda v. \bigsqcup \{ \widehat{[\pi]}\hat{d} \mid \pi \in \mathsf{P}(\mathsf{s}_G, v) \} \tag{3.3.2}$$

■

When showing that the MOP analysis is correct w.r.t. an analysis $\mathcal{B}$ working on $\hat{E}$, we can give an easy sufficient property that guarantees this if the domains of the two analyses are connected by a Galois Connection:

Theorem 3.3.3 (Correctness MOP I): Let the assumptions as stated in Definition 3.3.1 be valid and let furthermore $\hat{\mathcal{T}}'_{\cdot} : E \to \hat{E} \to \hat{E}$ be the transfer functions from Definition 3.2.13, $[\![\cdot]\!]'$ the path semantics from the same definition and $(\mathcal{P}(\Sigma), \alpha, \gamma, \hat{D})$ be a Galois connection. If the transfer functions $\hat{\mathcal{T}}_{\cdot}$ satisfy

$$\alpha \circ \mathcal{T}'_e \circ \gamma \sqsubseteq_{\to} \hat{\mathcal{T}}_e \tag{3.3.4}$$

Then the following property holds: $\alpha_{\to} \circ \mathcal{A}^{\mathsf{C}} \circ \gamma \sqsubseteq_{\to} \mathsf{MOP}$, i. e. MOP is correct w.r.t. $\mathcal{A}^{\mathsf{C}}$.

Proof: We first show for all $v \in V, \hat{d} \in \hat{D}, \pi \in \mathsf{P}(\mathsf{s}_G, v)$

$$\alpha([\![\pi]\!]'\gamma(\hat{d})) \sqsubseteq_{\hat{D}} \widehat{[\pi]}\hat{d} \tag{†}$$

by induction on π.

$\pi = \varepsilon$**:** Because of the Galois connection we have

$$\alpha(\gamma(\hat{d})) \sqsubseteq_{\hat{D}} \hat{d}$$

With $[\![\varepsilon]\!]' = \lambda S.S$ and $\widehat{[\varepsilon]} = \lambda \hat{d}.\hat{d}$ the claim follows.

$\pi = \pi'.e$**:** By the induction hypothesis we have

$$\alpha([\![\pi']\!]'\gamma(\hat{d})) \sqsubseteq_{\hat{D}} \widehat{[\pi']}\hat{d}$$

From this follows

$$\begin{aligned}
&\Rightarrow \ \hat{\mathcal{T}}_e(\alpha([\![\pi']\!]'\gamma(\hat{d}))) \sqsubseteq_{\hat{D}} \hat{\mathcal{T}}_e(\widehat{[\![\pi']\!]}\hat{d}) = \widehat{[\![\pi]\!]}\hat{d} \\
&\Rightarrow \ \alpha(\mathcal{T}'_e(\gamma(\alpha([\![\pi']\!]'\gamma(\hat{d}))))) \sqsubseteq_{\hat{D}} \widehat{[\![\pi]\!]}\hat{d} \\
&\Rightarrow \ \alpha(\mathcal{T}'_e([\![\pi']\!]'\gamma(\hat{d}))) = \alpha([\![\pi]\!]'\gamma(\hat{d})) \sqsubseteq_{\hat{D}} \widehat{[\![\pi]\!]}\hat{d}
\end{aligned}$$

Here the first step is true because of the monotonicity of the $\hat{\mathcal{T}}_.$. The second step is valid because of the assumed connection between $\mathcal{T}'_.$ and $\hat{\mathcal{T}}_.$. The third step is a consequence of the Galois connection.

We now have that, with $v \in V, \hat{d} \in \hat{D}, \pi \in \mathsf{P}(\mathsf{s}_G, v)(=: PP)$:

$$\begin{aligned}
&\quad \alpha([\![\pi]\!]'\gamma(\hat{d})) \sqsubseteq_{\hat{D}} \widehat{[\![\pi]\!]}\hat{d} \\
&\Rightarrow \ \bigsqcup\{\alpha([\![\pi]\!]'\gamma(\hat{d})) \mid \pi \in PP\} \sqsubseteq_{\hat{D}} \bigsqcup\{\widehat{[\![\pi]\!]}\hat{d} \mid \pi \in PP\} \\
&\Rightarrow \ \alpha(\bigcup_{\pi \in PP} [\![\pi]\!]'\gamma(\hat{d})) \sqsubseteq_{\hat{D}} \bigsqcup\{\widehat{[\![\pi]\!]}\hat{d} \mid \pi \in PP\} = \mathsf{MOP}(\hat{d})(v) \\
&\Rightarrow \ \alpha(\mathcal{A}^{\mathsf{C}}(\gamma(\hat{d}))(v)) \sqsubseteq_{\hat{D}} \mathsf{MOP}(\hat{d})(v) \\
&\Rightarrow \ \alpha_{\rightarrow} \circ \mathcal{A}^{\mathsf{C}} \circ \gamma \sqsubseteq_{\rightarrow} \mathsf{MOP}
\end{aligned}$$

where the second step can be concluded as a consequence of the distributivity of α (Lemma 3.2.22).

□

As before, we can choose the abstract transfer functions as $\hat{\mathcal{T}}_e = \alpha \circ \mathcal{T}'_e \circ \gamma$ and obtain the *best* MOP analysis possible under this Galois Connection.

As we have seen earlier, it is not always possible to define a Galois Connection for a given set $\hat{D}$ but it is possible to give a monotone concretization $\gamma : \hat{D} \rightarrow \mathcal{P}(\Sigma)$. This suffices to still assure the correctness of the analysis.

This approach has the disadvantage that the conditions on the transfer functions are a little bit more complicated and that we are not guided to the best possible version of the analysis.

Theorem 3.3.5 (Correctness MOP II): Let again the assumptions from Definition 3.3.1 be valid and let furthermore $\mathcal{T}'_. : E \rightarrow \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ be the transfer functions from Definition 3.2.13, $[\![\cdot]\!]'$ the path semantics from the same definition and $\hat{D}$ a complete lattice with a monotone function $\gamma : \hat{D} \rightarrow \mathcal{P}(\Sigma)$.

If the transfer functions $\hat{\mathcal{T}}_.$ satisfy

$$S \subseteq \gamma(\hat{d}) \Rightarrow \mathcal{T}'_e(S) \subseteq \gamma(\hat{\mathcal{T}}_e(\hat{d})) \tag{3.3.6}$$

then we have

$$\forall S, \hat{d} : S \subseteq \gamma(\hat{d}) \Rightarrow \forall v : \mathcal{A}^{\mathsf{C}}(S)(v) \subseteq \gamma(\mathsf{MOP}(\hat{d})(v))$$

i. e. MOP is a correct analysis on $\hat{D}$ w.r.t. $\mathcal{A}^{\mathsf{C}}$.

Proof:
First, we observe that the following property holds:

$$\bigcup\{\gamma(\hat{d}) \mid \hat{d} \in Y\} \subseteq \gamma(\bigsqcup Y) \tag{3.3.7}$$

This is because γ is monotone and

$$\begin{array}{rl} & \forall \hat{d} \in Y : \hat{d} \sqsubseteq_{\hat{D}} \bigsqcup Y \\ \Rightarrow & \forall \hat{d} \in Y : \gamma(\hat{d}) \subseteq \gamma(\bigsqcup Y) \\ \Rightarrow & \bigcup\{\gamma(\hat{d}) \mid \hat{d} \in Y\} \subseteq \gamma(\bigsqcup Y) \end{array}$$

We now show that

$$\forall \pi \in \mathsf{P}(\mathsf{s}_G, v), S, \hat{d} : S \subseteq \gamma(\hat{d}) \Rightarrow [\pi]' S \subseteq \gamma(\widehat{[\pi]}\,\hat{d})$$

by induction on π under the assumption $S \subseteq \gamma(\hat{d})$.

If $\pi = \varepsilon$, the claim is trivially true. For $\pi = \pi'.e$, we assume that our claim is valid for π':

$$\begin{array}{rl} & S \subseteq \gamma(m) \Rightarrow [\pi']' S \subseteq \gamma(\widehat{[\pi']}\, y) \\ \Rightarrow & \mathcal{T}_e'([\pi']' S) \subseteq \gamma(\hat{\mathcal{T}}_e(\widehat{[\pi']}\,\hat{d})) \\ \Rightarrow & [\pi]' S \subseteq \gamma(\widehat{[\pi]}\,\hat{d}) \end{array}$$

by using the assumptions on the transfer functions.

From this statement on the semantics of the paths to a node v we have

$$\begin{array}{rl} & [\pi]' S \subseteq \gamma(\widehat{[\pi]}\,\hat{d}) \\ \Rightarrow & \bigcup\{[\pi]' S \mid \pi \in \mathsf{P}(\mathsf{s}_G, v)\} \subseteq \bigcup\{\gamma(\widehat{[\pi]}\,\hat{d}) \mid \pi \in \mathsf{P}(\mathsf{s}_G, v)\} \\ \Rightarrow & \mathcal{A}^{\mathsf{C}}(S)(v) \subseteq \gamma(\bigsqcup\{\widehat{[\pi]}\,\hat{d} \mid \pi \in \mathsf{P}(\mathsf{s}_G, v)\}) \\ \Rightarrow & \mathcal{A}^{\mathsf{C}}(S)(v) \subseteq \gamma(\mathsf{MOP}(\hat{d})(v)) \end{array}$$

which proofs our claim. Here, the second step utilizes (3.3.7).

□

We can define the correctness of MOP on $\hat{D}$ not only w.r.t. the coarse analysis but also relative to another DFA, MOP' on a different domain $\hat{E}$ and with different abstract transfer functions $\hat{\mathcal{T}}_e'$. Then the correctness condition for a Galois connection between $\hat{E}$ and $\hat{D}$ is

$$\alpha \circ \hat{\mathcal{T}}_e' \circ \gamma \sqsubseteq_{\hat{D}} \hat{\mathcal{T}}_e$$

And in the general case the abstract transfer functions must satisfy

$$\hat{e} \sqsubseteq_{\hat{E}} \gamma(\hat{d}) \Rightarrow \hat{\mathcal{T}}'_e(\hat{e}) \sqsubseteq_{\hat{E}} \gamma(\hat{\mathcal{T}}_e(\hat{d}))$$

As said earlier, the MOP analyses are in general uncomputable as the least upper bound operator on the iteration results of infinitely many paths is in general uncomputable. Fortunately, one can formulate an approximation to the MOP analyses, which is computable, if the complete lattice $\hat{D}$ fulfills the *ascending chain condition*, i. e. every chain $\hat{d}_1 \sqsubset \hat{d}_2 \sqsubset \cdots$ is of finite length.

This second analysis, the *Minimal Fixed Point* analysis, works by computing a fixed point by iterating propagations of elements from $\hat{D}$ through the CFG, transformed by the abstract transfer functions:

Definition 3.3.8 (MFP-analysis): Let $\hat{\mathcal{T}}_{\cdot} : E \rightarrow \hat{D} \rightarrow \hat{D}$ be the monotone abstract transfer functions and $\hat{D}$ the lattice from Definition 3.3.1.

The MFP-analysis is defined as the smallest fixed point $\mathsf{MFP} : \hat{D} \rightarrow V \rightarrow \hat{D}$ of the equation system

$$\mathsf{MFP}(\hat{d})(v) = \begin{cases} \hat{d} & , v = \mathsf{s}_G \\ \bigsqcup\{\hat{\mathcal{T}}_{v' \rightarrow v}(\mathsf{MFP}(\hat{d})(v')) \mid v' \rightarrow v \in E\} & , \text{otherwise} \end{cases}$$

■

The following theorem is crucial:

Theorem 3.3.9 (Correctness MFP): There always exists a computable smallest fixed point of the MFP analyses and for all $\hat{d} \in \hat{D}$, $v \in V$ it holds that:

$$\mathsf{MOP} \sqsubseteq_{\rightarrow} \mathsf{MFP}$$

The MFP and MOP analyses are identical if the abstract transfer functions $\hat{\mathcal{T}}_{\cdot}$ are distributive.

Proof:
See [NNH99].

□

Although this theorem makes only an existential claim about the least fixed point of MFP, an algorithm to actually compute it can be derived from the definition of the MFP-analyses. Figure 3.3 shows the *work-set algorithm*. The algorithm begins with the starting value $\hat{d}$ at the start node and propagates the transformed values along the edges to other nodes until a stable assignment ($\mathcal{M}$) of nodes to values is reached. A central component of the algorithm is the work-set

W which contains all nodes which still have to be processed. In the beginning every node is assigned the value $\perp$, only the start node is assigned the starting value. The map $\mathcal{E}$ assigns to every edge the value of its source node, transformed by the transfer function of that edge. Initially, every edge is assigned the value $\perp$, except that the edges going out from s_G are assigned the value $\hat{\mathcal{T}}_e(\perp)$.

```
Input: CFG $G = (V, E)$, starting value $\hat{d} \in \hat{D}$, transfer functions $\hat{\mathcal{T}} : E \to \hat{D} \to \hat{D}$.
Output: a map $\mathcal{M} : V \to \hat{d}$ of nodes to values.
(* Init *)
$W$:=$V$; (* Working set *)
$\forall v \in V : \mathcal{M}(v)$:=$\perp$;
$\mathcal{M}(\mathsf{s}_G)$:=$\hat{d}$;
$\forall e \in E : \mathcal{E}(e)$:=$\perp$;
$\forall e = \mathsf{s}_G \to v \in E : \mathcal{E}(e)$:=$\hat{\mathcal{T}}_e(\perp)$
(* Iteration *)
while $W \neq \emptyset$ do
  $(v, W)$ :=extract($W$); (* extract a node *)
  new:=$\bigsqcup\{\mathcal{E}(v' \to v) \mid v' \to v \in E\}$;
  if ($\mathcal{M}(v) \neq$ new) then (* changed *)
    $\forall v \to v'' \in E$ :
      $\mathcal{E}(v \to v'')$:=$\hat{\mathcal{T}}_{v \to v''}$(new);
      $W$:=$W \cup \{v''\}$;
    $\mathcal{M}(v)$:=new;
  fi
od
```

Figure 3.3: Work-set algorithm

During the iteration a node from the work set W is selected; the value at this node is computed as the least upper bound of all the values at the incoming edges for this node. If the value has changed since the last iteration then the value at this node is set to the new value `new` and every outgoing edge $v \to v''$ is assigned the value $\hat{\mathcal{T}}_{v \to v''}(\texttt{new})$. The target nodes of the outgoing edges of this node are inserted into the work set.

The DFA we have presented here is usually referred to as *forward analysis* in the literature. Since we will only implement analyses that derive an approximation to a node map from an approximation to the starting values of the program, this suffices. One can also look at analyses that derive an approximation to the node map from an approximation to all *end values* of the program, i. e. the values at e_G. This can be done along the same ways as presented in the last two sections and leads to so-called *backward analyses*.

The work set algorithm shown above can be varied in many ways by using special heuristics to determine the node selected in every iteration, cf. [NNH99, Mar98, TMLA97].

3.3.1 Interprocedural Analyses

So far we have only looked at CFGs consisting of a single procedure (or function, subroutine). In reality every program contains several subroutines which are called from different contexts and with different arguments.

```
main()
{
  int j;
  ...
  p(j);
  ...
  q(1);
}
```

```
p(int k)
{
  int i;
  ...
  q(i+k);
}
q(int k)
{
  ...
}
```

Figure 3.4: Procedure calls

By context we mean the calling history of a procedure call. In Figure 3.4 procedure `q` can be called from `main` (with parameter `1`), or it can be called from `p` (which in turn was called from `main`), with parameter `i+k`. If we do not consider contexts in the DFA, then every call of a subroutine is handled like the combination of the different calls of this routine in the program, i. e. yields the least upper bound of the analysis results for each call. Especially when analyzing machine code, this can lead to a big loss in precision. E.g. a value analysis may be able to obtain precise contents of the stack pointer for each call of a subroutine, but the combination of these calls will give only a (large) interval of possible stack pointer values. Since local parameters are usually accessed via the stack pointer, this will also lead to a high loss for most data accesses in the subroutine. This imprecision will spill over to the pipeline analysis, since the data cache contents will not be known precisely enough. Thus, we need different analysis results for different contexts.

These different contexts can be taken into account in the analysis by several means, e. g.

- By *inlining* the bodies of the procedures. This only works for non-recursive procedures and furthermore leads to an exponential growth of the CFG with increasing call nesting depth.
- The data-flow value can be set to $\top$ at every procedure call. Although this is a safe approximation, one looses information and thus precision this way. The GNU C-Compiler `gcc` uses this technique in its analyses.
- One can compute *effects*, that is a mapping of incoming data-flow values to outgoing ones for every procedure.
- By using the *call string* approach, where the calling history is coded into the lattice of the data-flow values as a so-called “call string”, cf. [SP81].
- The *static call graph* method examines the static calling sequences of the program. For every such sequence (up to a certain length) the analysis is performed separately. Sequences that are longer than a fixed threshold are joined together and thus are less precise. This technique is implemented in the *Program Analyzer Generator* PAG, [Mar99b, TMLA97].

In the following we will only look at the last of the above techniques. We assume that procedure calls occur in the CFG as in Figure 3.5: every call consists of a **CALL** node that has an edge to the unique entry node of the procedure, its **ENTRY** node. A second node, the **RETURN** node at the calling site, also has an edge from the **CALL** node. This edge, which we call the **local** edge, serves the purpose of modeling the effects on local elements (variables, etc) with the associated transfer function. The unique end node of the procedure, its **EXIT** node also emits an edge to the **RETURN** node.

From a semantics point of view one must ask the question what relation the **local** edge has to the programs semantics: a control flow from the call of a procedure directly to the **RETURN** node is obviously impossible. One has to define the meaning of a path that contains such an edge, i. e. what the meaning of $\widehat{[\![\cdot]\!]}$ should be in this case.

A detailed theoretical foundation for this approach can be found in [Mar99b]. We will only present an intuitional argumentation: as soon as there are procedures

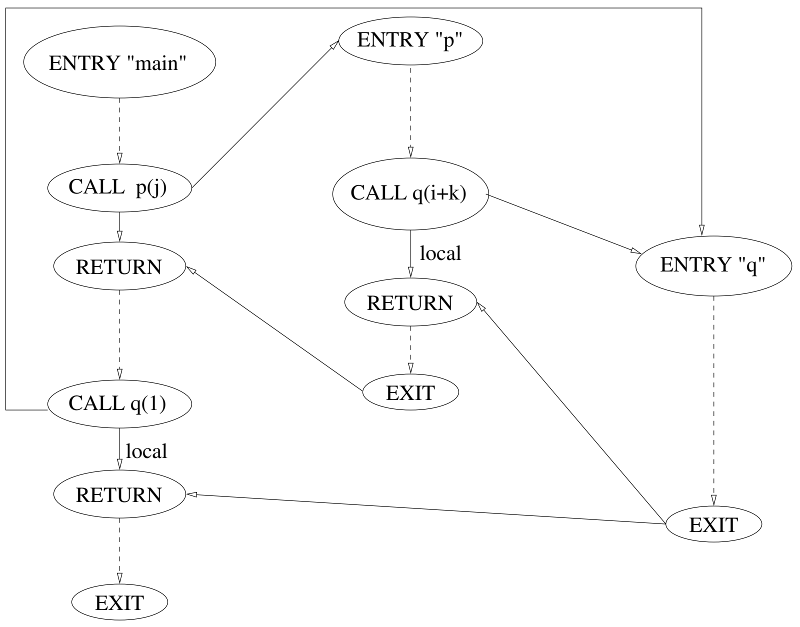


Figure 3.5: Interprocedural CFG

the problem of *incarnations* of procedure arises. That is, there can be several incarnations of a procedure. Each of these incarnations can have local variables which are not visible in other procedures. In a real implementation these incarnations are realized as stack frames (and a corresponding semantics can be given to them). The number of simultaneously active incarnations of a procedure is in general not bounded, so there can be no finite abstraction for this nesting, if one does not ignore local variables of procedures. So one has either the choice of leaving the effects of a procedure on the local variables of the calling procedure unmodeled[2]. In this case one has to set all information about local variables in the calling procedure at a call site to $\top$ to ensure correctness. Or one can keep the values of the local variables that are guaranteed to not be affected by the procedure call. The **local** edge serves exactly this purpose. It models the known effects on the local variables. By taking the join through the $\bigsqcup$ operator at the **RETURN** node this information is then correctly mixed with the information from the **EXIT** node of the called procedure.

[2]A procedure can also affect the variables not visible in the procedure by using pointers.

The technique of *static call graphs* can be implemented by using *mappings*: at every node in the CFG there is not one but several data-flow values, where each value corresponds to a different context at the same node. Now the edges connect the data-flow values at the nodes that have been connected by edges before. Through the type of mapping of the elements the static call graph technique can be realized. For the details cf. [TMLA97, Mar99a, Mar99b].

By this technique, the same work set algorithms can be used for interprocedural data-flow analyses as for intraprocedural analyses.

An even bigger problem for the precision of analysis information as computed by the value and pipeline analyses is the behavior of a program in its *loops*. Programs spend most of their execution time in some of the loops present in them. Thus, obtaining precise analysis results for loops will increase the precision of the analysis for the whole program. Loops are naturally a place where reuse of cache contents is most likely. Especially instruction cache but also data cache locality is very likely to occur in loops. Here, the first iteration of a loop usually loads the caches with the interesting memory blocks, e. g. the code of the loop itself. Further iterations will access this information in the cache and thus execute considerably faster. To capture this effect in an analysis of the execution, one must be able to distinguish loop iterations in the analysis. There are also cases, where one wants to distinguish not only the first loop iteration from the remaining ones but also the first N iterations from the remaining iterations. The VIVU approach implemented in PAG allows to do this. In this approach, a loop is virtually *inlined* and then virtually *unrolled* by factoring out loops as (virtual) *procedures* and reducing the problem to a mapping for procedure calls.

In Figure 3.6 an example loop together with the virtual transformation of its CFG is shown. Naturally, this transformation is only a conceptual one, the code of the program being analyzed is not altered. The transformation allows to gain information for separate iterations of the loop, thus increasing the precision of the DFA. On the left side of the figure is the original loop, on the right side the transformed CFG. A virtual procedure `loop_0000` with the code of the loop has been added. A call to this procedure is represented by the box labeled `loop_0000_first` in the original function `test_routine`. The iteration in the loop is modeled by a call to the loop, represented by the box labeled `loop_0000_rec` in the loop procedure.

Differentiating contexts in the DFA means that instead of one data flow element from $\hat{D}$ at every node of the CFG we have an array of elements, one for each context the node appears in. These elements are then connected following the CFG, but at **CALL** nodes the connection reflects the context change caused by the call according to the chosen mapping. This defines a *supergraph* upon which the propagation of data flow values is made, cf. [Mar99b, The02] for the details. In Figure 3.7 the loop from Figure 3.6 is shown with a mapping that differentiates

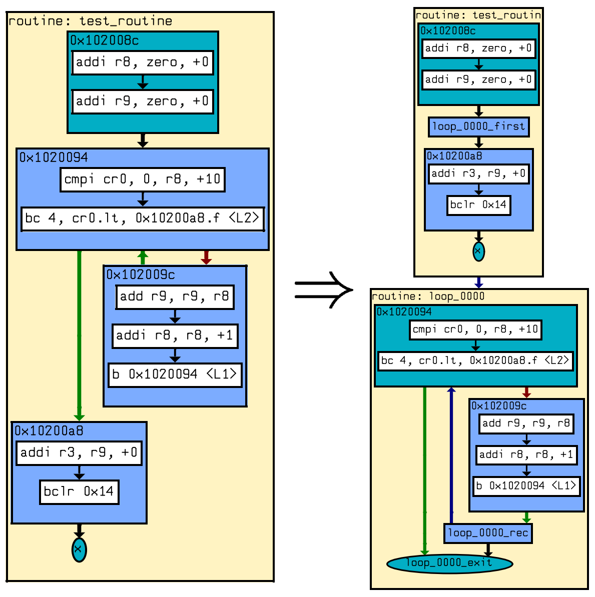


Figure 3.6: Program with loop and transformation

the first two loop iterations from the remaining ones. Thus, in the loop there are three data flow elements at every node. The elements with index 0 are those for the analysis of the first iteration of the loop, index 1 is for the second iteration and index 2 for all remaining iterations. The dotted lines in Figure 3.7 are the **local** edges between the **CALL** and **RETURN** nodes. The transfer functions at those edges and at the edges connecting the entry node elements with the first instruction node are the identity function $\lambda\hat{d}.\hat{d}$. The same is true for the edges from the **RETURN** to and from the **EXIT** nodes and the edges from the **CALL** nodes to the **ENTRY** nodes.

In the case that this loop can be reached from different places in the call graph, i. e. the subroutine containing this loop is called from more than one place, then additional contexts are added distinguishing between the various calls. As loops are treated in the same way as procedures, additional contexts also result from nested loops.

The precision that can be gained in the DFA by using separate contexts for separate loop iterations can be tuned against the time required for the analysis. In the extreme case, loops are unrolled by this technique up to their maximal iteration

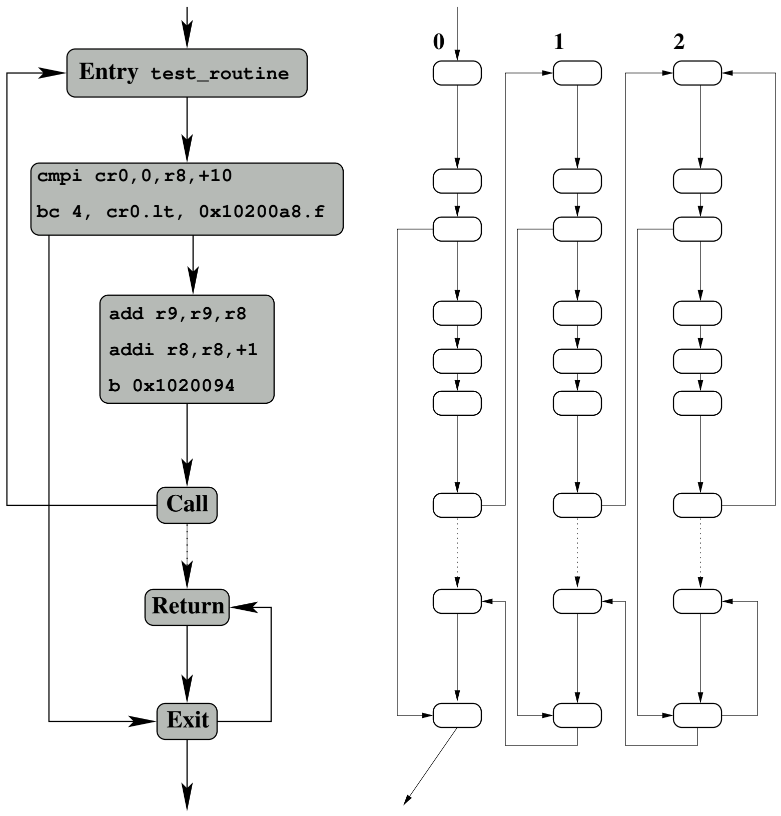


Figure 3.7: Mapping for two differentiated loop iterations

count, which must be known for WCET analysis anyhow. This gives the best precision but also the longest analysis time in general[3]. The number of contexts (i. e. separate data flow elements) increases rapidly for this complete unrolling of loops, especially in the presence of nested loops. How many loop iterations are unrolled can be tuned at run-time in the analysis. Some practical results about the precision gained and additional analysis time needed for different amounts of distinguished loop iterations are given in Section 6.5 for WCET analysis and in Section 8.1 for the effects on predictability of modern processors.

[3]Although there are cases, where the more precise analysis is faster, because it has to reiterate the same cycle in the CFG less often as the fixed point is reached earlier.

Chapter 4

Pipeline Modeling

Make for thyself a definition or description of the thing which is presented to thee, so as to see distinctly what kind of a thing it is in its substance, in its nudity, in its complete entirety, and tell thyself its proper name, and the names of the things of which it has been compounded, and into which it will be resolved.
Marcus Aurelius, Meditations, III/11

In order to judge the correctness of an analysis of the timing behavior we need a definition of the concrete behavior of the processor. In Chapter 5 we will develop correct abstractions and analyses for the concrete semantics using the techniques presented in Chapter 3.

We are interested in the timing behavior of a program being executed by the processor, i. e. we have to describe not only the results of the program execution (as far as they are relevant), but mainly how long the execution takes. There are several possibilities to choose from for the concrete semantics of the program execution:

- Simulation of low-level hardware models. Most chips nowadays are synthesized by translating low-level hardware models into gate layouts. The low-level models, written in a restricted set of VHDL ([VHD00, Ash02]) or Verilog ([TM91]), can be simulated too, and can thereby capture the timed execution of the program by simulating the processor.

- RTL level hardware models are VHDL or Verilog models at the register transfer level, which do not contain information necessary for synthesis.

Anyhow, they are still regarded as authoritative for the behavior of the processor on a cycle-exact level. These models are much faster to simulate and less complex than the low-level models used for synthesis.

- An effects semantics, defined in the classical way for instructions. This semantics defines the effects of program execution on a state by giving semantics to single instructions and defining how the semantics have to be combined to give the semantics for the whole program, e. g. by rules for sequencing instructions.
- Designing a higher level model that represents the effects of program execution on the state by hand, i. e. not based on an authoritative hardware model. Here "higher" means that the model is not constructed from the low level view of pins and signals in the processor but rather by identifying and modeling logical components and their interaction.

Each of these alternatives has its advantages and drawbacks:

- Low-level hardware models are too cluttered with unnecessary information, e. g. signal values that are not important for the timing of the program execution: from the 360 pins of the PPC 755, only 44 may be of interest for the timing of accesses to external memory. The situation would probably be much worse for internal signals. Also, it is difficult to obtain efficient abstractions from low-level models, because much of the structure has vanished at the level necessary for synthesis. Using simulation of these models is impractical due to the extremely slow simulation speed of full low-level models.
- RTL level models are at a higher level and thus are more efficient to simulate. However, they still contain much information that is never needed for timed program execution. From a RTL model one could slice out only those portions that influence program execution. This model would be much smaller and better to abstract. However, RTL models are rarely available for the designers of a timing analyzer as they are considered confidential information by the processor makers. In our work, no RTL models have been available and thus we could not follow this approach.
- The classical way to define semantics by combining smaller instruction semantics is difficult for modern processors, since here the instructions are executed in parallel, thus composition constructs and semantics for concurrent or parallel languages must be used. In addition, this parallelism is dynamic and the decision what instructions to execute in parallel is performed by the dispatching unit at run-time based on dynamic resource occupations. Thus,

the combination of the instruction semantics is not trivial at all. And it is not clear, how to easily obtain such a semantics and how to verify it. Furthermore, analysis of concurrent/parallel programs is still not very elaborated and easy to use *and* powerful methods are not available[1].

We choose to define a semantics along the lines of an RTL style hardware model, but with higher abstraction. This model has to be constructed from the available documentation about processor and system, augmented by experiments run on a real system. Its design can follow the overall design of the processor making it easier to obtain the model. Nonetheless, the validation of such a model is a non-trivial task and can be time consuming.

Since we are only interested in analyzing current processor designs, we restrict ourselves to *synchronous* designs, i. e. processors whose internal signals are synchronized against the rising and/or falling edge of a *system clock*. Although there are nowadays ideas to increase the speed of processors by using asynchronous logic, it has not been shown how the increased design complexity of this approach should be handled and how the processors could be verified.

Thus, we can handle time as *discrete*. The minimal time unit we want to use for measuring program execution is a full processor clock cycle, since instruction execution is normally synchronized against the rising edge of the processor clock. The minimum time interval that can have an observable effect is one half of a processor cycle[2].

Since the internal state of the processor and the contents of the main memory together are finite in size, it is natural to use the cycle-wise evolution of a *finite state automaton* as the concrete semantics of program execution. In this setting, program execution begins with a *state* that is set up such that the processor will start to execute the first instruction of the program (the program counter is set to the corresponding address). Then, the automaton will perform transitions, every transition taking one processor cycle, until the last instruction of the program has been completed. During each transition, the transition relation transfers the current state into a new state according to the (partial) execution of the current instruction. The total number of transitions made by the automaton gives the execution time of the program in processor cycles.

In the following section we will define the semantics of a program execution using the notations from Section 3.2 on the machine *program*, represented as a CFG G. We do this, because the analysis we are performing is a data-flow anal-

[1]Static analysis of concurrent programs in practice seems to be limited to either bit-vector problems, which are not useful in the context of WCET, or to the analysis of an interleaving-semantics, which is not practical for the size of our programs.

[2]It is one half of a cycle because internally events can be synchronized against the rising *and* the falling edge of the clock.

ysis on the program (or, more precisely, its CFG). Section 4.2 describes how a processor model can be obtained in a way that reduces design complexity by a series of abstractions.

4.1 Finite State Automata

For a given processor its execution can be described by giving a set $\mathcal{S}$ of *(machine) states* that the processor may be in. Such a state not only represents the inner components of the processor but also all peripheral hardware states of the system, e. g. the contents of main memory or the inner state of the main chipset connecting the processor to the rest of the system. Apart from the set $\mathcal{S}$ there is a *transition function* $\mathbb{T}$ between states. To denote the fact that the program execution has halted, we introduce a special state $\perp$, so that $\mathbb{T} : \mathcal{S} \to \mathcal{S} \cup \{\perp\}$. A transition from a state $s \in \mathcal{S}$ to a state $\mathbb{T}(s)$ takes exactly one processor cycle. To summarize, we define

Definition 4.1.1 (Finite Automaton): A finite automaton A is a pair $(\mathcal{S}, \mathbb{T})$ where $\mathcal{S}$ is a finite set of states and $\mathbb{T} : \mathcal{S} \to \mathcal{S} \cup \{\perp\}$ is a transition function between states. When the automaton is in state $\perp$, we say that it has halted.
■

Naturally, by inspecting a state $s \in \mathcal{S}$, we must be able to say which of the possibly many instructions being executed in parallel is the oldest one (regarding program order) and if an instruction has been retired. How this is done depends on the processor and the model, but the information is present in some way in the *components* (see below) of a state. We will introduce generic *state predicates*, which must be defined for each automaton, that given a state return the necessary information.

In order to argue about properties of (short sequences of) instructions we will describe a (machine) program G by its CFG, as in Section 3.2. Each node of the CFG represents one machine instruction and is labeled with the address of that instruction, denoted by $\mathsf{addr}(n)$ and the instruction itself, denoted by $\mathsf{prog}(n)$. In the case of a conditional branch instruction, the edges going out of that node are labeled with T and F for the edge corresponding to the taken branch and the fall-through edge, resp. Given a program G, the execution of the finite automaton starts with a state $\mathsf{ss}_G(d)$, which is set up such that the processor will start to fetch the instruction at $\mathsf{addr}(\mathsf{s}_G)$. Here d denotes the input data for this particular run, which come from a set D.

It should be noted that the program we are going to execute is present in the state, as it is stored in memory and the memory contents are part of the state. Thus, by defining $\mathcal{S}$ to be the set of all possible states, we have defined the finite automaton for all possible programs, so that the automaton is only dependent on the processor and the model itself that gives the structure of the states and the transition function.

The execution of the program by the processor can be viewed in two ways:

- By giving the trace of the states that occur during execution of the finite automaton A.
- By giving the path π through the programs CFG (as in Definition 3.1.4) together with transfer functions for edges in the CFG (cf. Definition 3.1.3).

The first view is more natural in that it represents the real execution of a processor more closely, while the other one makes is easier to argue about properties during execution of a given instruction and the correctness of an analysis. Anyhow, one view is sufficient to define the other. We will start by defining the state trace first, then the execution-path semantics next, which will be used later to prove the correctness of the resulting analyses.

As noted in Section 3.1 we can define a system-centric semantics by just looking at the evolution of a system as a whole. For an automaton we can define the cycle-wise evolution by a sequence of states called the *state trace* of processor execution starting from a state s:

Definition 4.1.2 (State Trace): Given a finite automaton $A = (\mathcal{S}, \mathbb{T})$, the state trace starting from state $s \in \mathcal{S}$ is defined by

$$T_A(s) = \begin{cases} \varepsilon & \text{, if } \mathbb{T}(s) = \bot \\ s.T_A(\mathbb{T}(s)) & \text{, otherwise} \end{cases}$$

■

Since we are requiring that every program execution terminates, the transition function $\mathbb{T}$ must not produce infinite chains of states. I.e. every trace has a finite length. The length of a trace is the number of cycles the execution took, thus giving the execution time. The execution of the whole program G with input d is represented by $T_A(\mathsf{ss}_G(d))$.

In Section 3.1 we already argued that for a modular analysis one needs to go away from the system-centric view to the *instruction-centric* view that allows to associate partial state traces to instructions, thus defining the execution of an instruction. As noted earlier, it is processor dependent when an instruction has finished execution. Some parts of the processor states will reflect this information, either a specific part alone or several parts in conjunction. To generalize these specific dependencies, we introduce predicates that describe the information of a state w.r.t. the execution of one instruction. These predicates have to be defined for every processor and its accompanying automaton. In Sections 4.3 and 4.4 we will give examples of the definition of these predicates using the processor modeling introduced later.

So, to argue about the execution of single instructions (associated with nodes n in the CFG), we introduce the following predicates and functions:

Definition 4.1.3 (State Predicates): The predicate $\mathcal{F}(n,s)$ is true, if the instruction at node n is finished in state s. The predicate $\mathcal{N}(n,s)$ is true, if the instruction at node n is the next instruction to be finished in state s after the current one. The predicate $\mathcal{H}(s)$ is true if program execution has halted in state s. The function $\mathcal{R}(n,s)$ returns for a state s, in which the instruction at n is finished, a state s' where the acknowledge of n being finished has been removed. Thus, the second oldest instruction being executed in s is the oldest being executed in s'. ■

4.1.1 The Meaning of State Predicates

These state predicates allow us to *conceptually* reduce the overlapped, parallel execution of multiple execution in the pipeline to a sequential execution. This is similar to approaches taken in the area of the verification of processor designs, e. g. [BD94, HYHD95, JSD98, SJD98, DP97, Bur96]. There, the problem is to verify that an implementation of a processor design is correct w.r.t. a *sequential* specification (ISA[3]). The ISA defines the (observable) effects of instructions on the programmer visible system state (registers and memory). In it, instruction execution is conceptually sequential, as in our instruction-centric view. The approaches taken in verifying an actual implementation also require to establish a connection between the processor states in the actual execution and the effects of instructions on the state as defined in the ISA. This is either done by artificially flushing an implementation pipeline state and the comparing the ISA-observable parts of it against the result of an instructions execution in the ISA or by defining a refinement relation between pipeline states of different implementations (where one implementation is the sequential ISA) with the help of an abstraction and observable components of pipeline states.

Our state predicates differ from these approaches in that they don't talk about instruction effects on some programmer visible part of the system but rather determine when an instruction can have no further influence because it has left the pipeline and which instruction should be considered the next one to execute in a sequential view. E.g. in processor verification an instruction i may still be in the pipeline in some retirement buffer in a state s, but all of its effects have been committed. For verification, this state is not distinguishable (and need not be) from a

[3]Instruction Set Architecture

state s' where the instruction has left the pipeline completely, as they are equal after projection to the ISA-observable parts. For our state predicates, $\mathcal{F}(i,s)$ would be false, while $\mathcal{F}(i,s')$ would be true. Thus, the states are considered not equal under timing-aspects of the execution.

Two Examples for State Predicates

To illustrate this concept further, we give two examples how state predicates can be defined with the help of special components in the state. The first example is a simple DLX like pipeline without super-scalarity of out-of-order execution[4]. The second one features a branch prediction that folds away branches in the fetch stage.

IF	ID	EX	WB

Figure 4.1: Simple Pipeline

Consider the simple pipeline in Fig. 4.1, which consists of an instruction fetch (IF), a decode (ID), an execution (EX) and write-back (WB) stage.

As a first example consider that all instructions flow sequentially through the pipeline, branches are computed in the EX stage, redirecting the fetching. Among other components, a pipeline state s records for each state, which instruction i is in the stage at the moment. E.g., we write $s(\mathrm{WB})$ for the instruction in the WB stage in state s (which can be ε if the stage is empty). To record which instruction has left the WB stage and thus the pipeline, we have another component RT (retired) in the state, holding the last retired instruction. Naturally, the components are updated together with other components (not shown in the example) during one application of $\mathbb{T}(s)$ to model the execution flow.

With this we can define the state predicates in the following way:

$$\begin{aligned}
\mathcal{F}^1(i,s) &= \begin{cases} \text{true} & \text{, if } s(\mathrm{RT}) = i \\ \text{false} & \text{, otherwise} \end{cases} \\
\mathcal{N}^1(i,s) &= \begin{cases} s(\mathrm{WB}) & \text{, if } s(\mathrm{WB}) \neq \varepsilon \\ s(\mathrm{E}) & \text{, if } s(\mathrm{WB}) = \varepsilon \wedge s(\mathrm{E}) \neq \varepsilon \\ s(\mathrm{D}) & \text{, if } s(\mathrm{WB}) = \varepsilon \wedge s(\mathrm{E}) = \varepsilon \wedge s(\mathrm{D}) \neq \varepsilon \\ s(\mathrm{IF}) & \text{, otherwise} \end{cases} \\
\mathcal{H}^1(s) &= \begin{cases} \text{true} & \text{, if } s(\mathrm{IF}) = s(\mathrm{D}) = s(\mathrm{EX}) = s(\mathrm{WB}) = \varepsilon \\ \text{false} & \text{, otherwise} \end{cases} \\
\mathcal{R}^1(i,s) &= s[\mathrm{RT} \mapsto \varepsilon]
\end{aligned}$$

[4]For simplicity, we leave out the MEM stage which performs memory accesses.

Thus, an instruction has left the pipeline if it occurs in RT. The next instruction is the one in the last non-empty stage. Execution has halted if there is no instruction being executed or fetched. The $\mathcal{R}$ predicate simply clears the RT component of the last finished instruction.

For the second example, we take the same pipeline layout but now unconditional branches are folded in the ID stage, i. e. they are removed from the instruction stream and never enter the EX or WB stages. When waiting for such a branch to finish execution (i. e. removal from the pipeline), there can be several scenarios:

- The branch has not yet been fetched, thus it has not reached the ID stage. Then we must perform more cycle transitions until it is discarded in the ID stage after fetching completes.
- It has already been discarded while we waited for the end of a predecessor instruction. Then its execution time is zero.
- It is in the IF or ID stage but has not yet been discarded.

To keep track of this, we add another component to the state, NFB (number of folded branches). NFB is a counter that records the number of branches that have been folded out in the ID stage. Is is updated during $\mathbb{T}(s)$ if a branch is folded out in the ID stage.

With the help of three functions br, cb, and targ, where $\text{br}(i)$ is true if instruction i is an unconditional branch; $\text{targ}(i)$ is the target instruction of an unconditional branch i, and $\text{cb}(i)$ is true if the instruction i is a conditional or computed branch, we can define the state predicates by

$$\begin{aligned}
\mathcal{F}^2(i,s) &= \begin{cases} \text{true} & \text{, if } s(\text{RT}) = i \wedge \neg\text{br}(i) \\ \text{true} & \text{, if } \text{br}(i) \wedge s(\text{NFB}) > 0 \\ \text{false} & \text{, otherwise} \end{cases} \\
\mathcal{N}^2(i,s) &= \begin{cases} \text{targ}(i) & \text{, if } \text{br}(i) \\ \mathcal{N}^1(i,s) & \text{, if } \neg\text{br}(i) \wedge \text{cb}(i) \\ i+4 & \text{, otherwise} \end{cases} \\
\mathcal{H}^2(s) &= \mathcal{H}^1(s) \wedge s(\text{NFB}) = 0 \\
\mathcal{R}^2(i,s) &= \begin{cases} s[\text{RT} \mapsto \varepsilon] & \text{, if } \neg\text{br}(i) \\ s[\text{NFB} \mapsto s(\text{NFB}) - 1] & \text{, otherwise} \end{cases}
\end{aligned}$$

Here, an unconditional branch is finished if it has been folded out. If the branch has not yet reached the ID stage, then $s(\text{NFB})$ is zero and more applications of $\mathbb{T}$ (cycles) must pass before it reached the ID stage and NFB is incremented. The $\mathcal{R}$ predicate now has to differentiate between unconditional branches and other instructions (only the latter end up in RT). The next instruction also becomes more difficult to determine, as one has to conceptually reinsert the discarded branches. Thus, the $\mathcal{N}$ predicate differentiates between instructions whose successor is already in the pipeline or statically known (non-branches and computed branches, resp.) and folded branches as the result. The folded branches appear either as $i+4$ (i. e. the next instruction after i) or as the statically known targets of an unconditional branch to another such branch.

4.1.2 Instruction Execution Semantics

The instruction-centric semantics in Definition 3.1.4 is defined using the transfer functions for each edge in the program. Our set Σ is defined by $\Sigma = \mathcal{S} \times \mathcal{Z}$ as the set of pairs containing a machine state and the number of cycles the execution took so far[5].

We then define the transfer functions as follows:

Definition 4.1.4 (State Transfer Functions): Let $A = (\mathcal{S}, \mathbb{T})$ be a finite automaton and let G be a program. We define the **state transfer functions** $\mathcal{T}_{n \to n'} : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ as

$$\mathcal{T}_{n \to n'}(t) = \begin{cases} \emptyset & \text{, if } t = \{(s,m)\} \land \mathcal{F}(n,s) \land \\ & \quad \neg\mathcal{N}(n',s) \\ \{(\mathcal{R}(n,s),m)\} & \text{, if } t = \{(s,m)\} \land \mathcal{F}(n,s) \land \\ & \quad \mathcal{N}(n',s) \\ \mathcal{T}_{n \to n'}(\{(\mathbb{T}(s), m+1)\}) & \text{, if } t = \{(s,m)\} \land \neg\mathcal{F}(n,s) \\ \emptyset & \text{, otherwise} \end{cases}$$

■

This captures the fact that we define the execution of an instruction to have finished if it *leaves* the pipeline. The number of state transitions with states that still execute this instruction is the number of cycles it takes to finish. Note that normally the execution of this instruction has already been started earlier. Thus, it can happen that it takes zero cycles to complete, if it has already been retired (which can happen in the case of multiple retirements, as in the Motorola PPC 755). For an architecture with out-of-order retirement, when an instruction i retires before its predecessor (in fetch or program order) i', then also i will have an execution time of 0, as it has already left the pipeline. The execution of i has occurred in parallel with that of i' and as i' takes longer and we have waited for a number of cycles for it to leave the pipeline, we need not assign any real execution time to i later (in fetch order).

[5] $\mathcal{Z} = \{0, 1, \ldots, T_{\max}\}$ is a subset of $\mathbb{N}$, containing all possible execution times, cf. Chapter 5.

Note that if an instruction determines that it will not execute along the edge $n \rightarrow n'$ (but rather along an edge $n \rightarrow n''$), then the result of $\mathcal{T}_{n \rightarrow n'}(t) = \emptyset$ meaning infeasible execution along this edge.

By the definition of the transfer function, the number m of execution cycles so far does not influence the new state. Thus, if $\mathcal{T}_e(\{(s,m)\}) = \{(s', m+k)\}$ then we have $\{(s',k)\} = \mathcal{T}_e(\{(s,0)\})$. This allows us later to have an abstraction for the execution of one instruction starting at time 0 that describes every execution of the instruction (starting from the same state s) at an arbitrary time m.

Two Examples for Instruction Execution

Again, we give two examples for the execution of an instruction i, with the same two simple processors as above.

i_1	
i: ba l	
i_2	$\vdots$
i_3	l: j_1
i_4	j_2
$\vdots$	j_3
	$\vdots$

Figure 4.2: An example instruction sequence

Consider that we are executing the simple program in Fig. 4.2 and are looking at $\mathcal{T}_{i_1 \rightarrow i}(\{(s,0)\})$ for a state s, i. e. the execution of instruction i_1.

For the first simple processor architecture, the instruction i_1 is finished when it ends up in the RT component. The same holds for the unconditional branch i.

Assuming that i is in the IF stage in state s and i_1 is in the ID stage and no other instruction is being executed, the execution proceeds in the following way:

Cycle	IF	ID	EX	WB	RT	$\mathcal{F}^1$	$\mathcal{N}^1$
$0(s)$	i	i_1	ε	ε	ε		
1	i_2	i	i_1	ε	ε		
2	i_3	i_2	i	i_1	ε		
$3(s^1)$	j_1	ε	ε	i	i_1	i_1	i
$3(s^2 = \mathcal{R}(i_1, s^1))$	j_1	ε	ε	i	ε		
$4(s^3)$	j_2	j_1	ε	ε	i	i	j_1
$4(s^4 = \mathcal{R}(i, s^3))$	j_2	j_1	ε	ε	ε		

In this table, the last two columns give the instructions j for which $\mathcal{F}^1(j,s)$ holds and which is the next instruction for that instruction ($\mathcal{N}^1$).

After the 3rd application of $\mathbb{T}$ to the start state s, instruction i_1 has left the pipeline. Thus, for instruction i_1 we have $\mathcal{T}_{i_1\to i}(\{(s,0)\}) = \{(s^2,3)\}$. Here, s^2 is the result of using $\mathcal{R}$ on the state represented by the previous line in the above table (s^1).

Going on from here, we have that $\mathcal{T}_{i\to j_1}(\{(s^2,3)\}) = \{(s^3,4)\}$, as the branch i will retire in 1 more cycle.

For the processor variant that folds out branches in the ID stage, the execution from an initial start state proceeds in the following way:

Cycle	IF	ID	EX	WB	NFB	RT	$\mathcal{F}^2$	$\mathcal{N}^2$
$0(s)$	i	i_1	ε	ε	0	ε		
1	i_2	i	i_1	ε	0	ε		
2	j_1	ε	ε	i_1	1	ε		
$3(s^1)$	j_2	j_1	ε	ε	1	i_1	i_1	i
$3(s^2 = \mathcal{R}(i_1,s^1))$	j_2	j_1	ε	ε	1	ε	i	j_1
$3(s^3 = \mathcal{R}(i,s^2))$	j_2	j_1	ε	ε	0	ε		

Thus, i_1 again finishes in 3 cycles but in this sequence, i has already been folded out (cycle 2). After $\mathcal{T}_{i_1\to i}(\{(s,0)\}) = \{(s^2,3)\}$ we have $\mathcal{T}_{i\to j_1}(\{(s^2,3)\}) = \{(s^3,3)\}$ as i has already left the pipeline as indicated by $\mathcal{F}^1(i,s^2)$ being true. One can say that i takes zero cycles to execute (from start state s^2 on).

With this architecture, the sequence i_1, i will finish after 3 cycles, one cycle earlier compared to the first processor without branch folding.

Collecting Semantics

Our primary point of interest is not the whole set of states that can occur at the end of a program as result of its execution but only the part that describes the *time* it took to execute the program. And as we are interested in an upper bound for *any* execution of a program with arbitrary input, we will look at the *collecting semantics* (cf. Definition 3.1.8)

$$\mathcal{A}_G^{\mathsf{co}}(\{(\mathsf{ss}_G(d),0) \mid d \in D\}) \tag{4.1.5}$$

Here, D contains all possible input data. By looking at the final instruction of the program, we can then define the WCET of the program as the maximum number of cycles of every execution:

$$\mathrm{WCET}(G) = \max\{n \mid (s,n) \in \mathcal{A}_G^{\mathsf{co}}(\{(\mathsf{ss}_G(d),0) \mid d \in D\})(\mathsf{e}_G)\} \tag{4.1.6}$$

As explained in Section 3.2.1, we can utilize the less precise Coarse analysis $\mathcal{A}_G^{\mathsf{C}}$ as a basis for our analyses in later chapters instead of $\mathcal{A}^{\mathsf{CO}}$. The WCET defined via the Coarse Analysis is $\max\{n \mid (s,n) \in \mathcal{A}_G^{\mathsf{C}}(\{(\mathsf{ss}_G(d),0) \mid d \in D\})(\mathsf{e}_G)\}$, which is greater or equal to $\mathrm{WCET}(G)$.

4.1.3 Inputs to Finite Automata

The finite automaton of Definition 4.1.1 is intended to model the execution of a real computer system, more specifically, a hard-real time system. One may wonder why there is no notion of the inputs to the system (e. g. sensor readings, etc) in the definition of the automaton *execution* ((4.1.2) and via the transfer functions in (4.1.4)). As stated briefly at the introduction of the start state on page 106 and in equations (4.1.5, 4.1.6), we encode all input data to the program in the start state $\mathsf{ss}_G(d)$. This encoding can be represented as the finite vector of sensor readings that occur during execution. Naturally, this vector is not part of the physical computer system. By putting the input vector into the state, we avoid having to deal with an extra input vector in the definition of the automaton execution, without losing any possible finite execution.

Since we require that termination of any program under consideration is guaranteed, there can be only finitely many readings of input data. Thus, the state space with the embedded input data vector is still finite. This would not be the case, if we allow arbitrary (i. e. including non-terminating) programs. In addition, since we assume that the program is already somehow represented in the state (i. e. in computer memory), there can be only finitely many programs. Thus, also the set of possible input data D can be assumed to be finite.

4.2 A Sequence of Abstractions

While the last section was concerned with defining the semantics of a given finite automaton, i. e. with given set $\mathcal{S}$ and transition function $\mathbb{T}$, this section will describe how to define the structure of the elements of $\mathcal{S}$ (the states) and how to give the transition function $\mathbb{T}$. Since the inner workings of a processor can be quite complex, especially if the processor has features like speculative execution or out-of-order execution, one must find a way to describe the state and state transitions in a modular way. Otherwise, the complexity of the dependencies in the state transitions would be impossible to handle. As stated before, we will adopt a high-level hardware modeling concept to ease the definition of the state and transition function.

4.2.1 Introducing Components and Units

First, a state has an inner structure; we can view a state as a tuple of *components*. A component might represent the contents of a register or a prefetch queue, etc. Naturally, the exact number and types of components depends on the processor to be described.

Another feature that is very common in the description of hardware is that there is *delayed* information flow, i. e. new values of components take effect only in the *next* cycle of the execution, see Figure 4.3 on page 115 for an example involving a "`clear`" signal and a "`result`" output signal of a digital circuit.

Assume we have a trigger signal `clear`, a result signal `result`, and an integral state variable `arg`.
The `clear` signal should take effect only in the next execution of the code, i. e. `result` should only be set in the next cycle.

```
if (arg>10)
   clear:=1
if (clear==1)
   result:=0
```

does not work since `result` is set immediately. Yet, we can rewrite this example to use a *latch* `clear_delayed` that is copied at the end of the cycle.

```
if (arg>10)
   clear_delayed:=1
if (clear==1)
   result:=0
   ⋮         ⋮
clear:=clear_delayed
```

Figure 4.3: Delayed signal example

We call a component of a state that can be viewed as such delayed data a *delayed signal*, borrowing from the semantics of signal assignments with delta delays in VHDL, which exhibit the same behavior as delayed signals.

So far, a state is:

Definition 4.2.1 (State): A state μ is a tuple of components and delayed signals. The delayed signals are represented by a mapping

$$\{n \mapsto d_n \mid n \in \text{Lab}^{\delta}, d_n \in D_n\}$$

the components by a mapping

$$\{m \mapsto c_m \mid m \in \text{Lab}^{c}, c_m \in C_m\}$$

The set Lab^{δ} is a set of *names* for the delayed signals; likewise Lab^{c} is a set of names for the components. D_n and C_n are the domains of the delayed signals and components. So a state μ can be written as the union of a mapping for the delayed signals and the components:

$$\mu = \{n_1 \mapsto d_1, \dots, n_k \mapsto d_k, m_1 \mapsto c_1, \dots, m_l \mapsto c_l\}$$

with $n_i \in \text{Lab}^{\delta}, d_i \in D_{n_i}, m_i \in \text{Lab}^{c}, c_i \in C_{m_i}$.

Each delayed signal has a default value, represented by a function $\delta_{\perp} : \text{Lab}^{\delta} \to \bigcup_{n \in \text{Lab}^{\delta}} D_n$ which gives the value of a delayed signal, if it is not explicitly asserted. We write $[\mu]$ for the set containing all possible states and $[\delta]$ for all possible mappings for delayed signals. ■

A state represented by a mapping μ can equivalently be represented by a tuple from $D_{m_1} \times \cdots \times D_{m_l} \times C_{n_1} \times \cdots \times C_{n_k}$. We can switch between both representations via a bijection $\psi : \{1, \dots, k+l\} \to \{n_i \mid 1 \leq i \leq k\} \cup \{m_i \mid 1 \leq i \leq l\}$. However, we will introduce names for objects in our model description language later anyhow. To ease the definition of the semantics of that language, we will use names already here.

The values of the delayed signals and the components in a state s are changed by the transition function $\mathbb{T}$ based on the actual values:

$$\mu^{\text{next}} = \mathbb{T}(\mu^{\text{actual}})$$

Clearly, not all $\mu(n)^{\text{actual}}$ are needed for the computation of a $\mu(m)^{\text{next}}$. From this, we can deduce that it may be a good idea to *group* components that represent information belonging together with little influence on other components. E.g. all components that directly represent the state of branch prediction of the processor may be grouped together. We call a set of grouped components a *unit*. Now we make explicit the dependencies between old and new values of components in different units by defining *instantaneous signals*. Such signals, which can carry parameters, like in VHDL, are sent from one unit to another unit. Issuing a signal carries the information about the inner state of a unit to other units. With this, the update of the inner state of a unit only depends on the delayed signals, its inner state and the instantaneous signals it received. The instantaneous signals are also used to structure the state evolution: they denote more abstract *events* that happen. As an extension, there can be units that represent *global register files* that are visible to all units (although this can be modeled by using instantaneous signals broadcasting the unit state to all other units).

Definition 4.2.2 (Unit): A unit u is a collection of components, represented as a mapping

$$\mu_u = \{n \mapsto c_n \mid n \in \text{Lab}^c_u, c_n \in C_n\}$$

where $\text{Lab}^c_u \subseteq \text{Lab}^c$ and we decompose the components of a state, μ, into a tuple $(\mu_1, \dots, \mu_N)$ of units, such that $\text{Lab}^c_u \cap \text{Lab}^c_{u'} = \emptyset \iff u \neq u'$ and $\bigcup\{\text{Lab}^c_u \mid 1 \leq u \leq N\} = \text{Lab}^c$. Thus we have N units $\mu_1, \dots, \mu_N$ whose union contains all the components from μ and we can write $\mu = \bigcup_{1 \leq i \leq N} \mu_i \cup \delta$, where δ are the delayed signals of the state. We write $[\mu_i]$ for the set containing all possible contents of unit μ_i. ■

A named instantaneous signal ι is sent from one unit to one or more units carrying data from a domain I_ι. Emitting and receiving an instantaneous signal will simply be done by adding an instance of the signal to the set of asserted signals, as we will later see.

Definition 4.2.3 (Instantaneous Signal): An instantaneous signal ι is a tuple $(u, \{u_1, \dots u_n\}, l, I_\iota)$, where u and $u_1, \dots, u_n$ are units, $l \in \text{Lab}^\iota$ is a label and I_ι is the domain of the signal. An instance of a signal is a pair (l, i), where $i \in I_\iota$ is the actual data carried by the signal. u is the unit emitting the signal, $u_1, \dots, u_n$ the ones receiving it, $\forall i : u \neq u_i$. An instantaneous signal with label l can be sent from only one unit but be received by multiple units. Thus, for all signals (u, U, l, I_ι) there must not be a different one $(u', U', l', I_{\iota'})$ with $l = l'$. We write $[[\iota]]$ for the set of all instantaneous signals ι. We write $[\iota]$ for the set containing all possible instances of instantaneous signals, i. e. for the set $\{(l, i) \mid (u, U, l, I) \in [[\iota]] \wedge i \in I\}$. A set I of instantaneous signals, $I \subseteq [\iota]$ with $(l, v) \in I \Rightarrow \nexists (l, v') \in I : v \neq v'$ will be written as a partial mapping

$$I = \{n_1 \mapsto i_1, \dots, n_l \mapsto i_l\}$$

where $n_j \in \text{Lab}^\iota$ and $i_j \in I_j$. Note that $\{n_1, \dots, n_l\}$ need not be equal to Lab^ι, i. e. the mapping is partial. For an $l \in \text{Lab}^\iota$ we write $\text{targ}(l)$ for the set U of a signal $(u, U, l, I) \in [[\iota]]$. ■

Unit components and delayed signals together with asserted instantaneous signals can be written as a mapping $\nu = \mu \cup I$. We write $[\nu]$ for the set of all combined mappings. Note that $[\mu] \subseteq [\nu]$.

For a mapping f from $[\nu]$ and a set N we write $f \downarrow N$ for the mapping obtained from f by containing only labels from N and $f \uparrow N$ for the mapping equal to f but not containing any labels from N. To denote the update of a mapping f for label l with new value v, we write $f[l \mapsto v]$. For two mappings f and $g = \{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\}$ we write $f \oplus g$ for the mapping

$$f[l_1 \mapsto v_1] \cdots [l_n \mapsto v_n]$$

.

Our notation for updating units will be sequential, i. e. one unit will be updated after the other. Since the instantaneous signals give the dependencies of a unit's new state w.r.t. other units, the updates must be done such that all units that emit a signal must be updated before a unit that receives that signal. This is not necessary for delayed signals. Also, we assume that signals emitted from different units but received by the same unit are distinct. This can easily be made explicit by using different labels (possibly including the sending units name) for the signals. This is also true for delayed signals: only one unit may change the value of a delayed signal. And again, this can be circumvented by using more delayed signals, parameterized by which unit changed the signal. Note that this implies that during the state update of a unit, the update must decide which signal from which unit takes precedence. We could circumvent this by introducing a form of *resolved signals* as is done in VHDL to decide a signals value if it is driven by multiple sources, cf. [VHD00, clause 2.4]. We choose not to, since the priority issues showed themselves as being easy to resolve.

Given the set $[[\iota]]$ we can define a *dependency graph* of the units connected by the instantaneous signals:

Definition 4.2.4 (Unit Dependency): Given the set of instantaneous signals, $[[\iota]]$, we define the dependency graph $\mathbb{D} = (V_{\mathbb{D}}, E_{\mathbb{D}})$ by

$$\begin{array}{rcl} V_{\mathbb{D}} & = & \{1, \ldots, N\} \\ E_{\mathbb{D}} & = & \{(u, u') \mid (u, U, l, I) \in [[\iota]] \wedge u' \in U\} \end{array}$$

We require that $\mathbb{D}$ is acyclic, i. e. there is no cyclic dependency by instantaneous signals among units.

Given $\mathbb{D}$, let $\prec_{\mathbb{D}} \subseteq \{1, \ldots, N\} \times \{1, \ldots, N\}$ be an ordering on units that is asymmetric and irreflexive such that

$$\exists (u, u') \in E_{\mathbb{D}} \Rightarrow u \prec_{\mathbb{D}} u'$$

We write $\prec_{\mathbb{D}}\downarrow U'$ for the restriction of $\prec_{\mathbb{D}}$ to $U' \times U'$. By $\min_{\prec_{\mathbb{D}}}(U')$ we denote an $u \in U'$ that is minimal, i. e. $\nexists u' \in U' : u' \prec_{\mathbb{D}} \min_{\prec_{\mathbb{D}}}(U')$. If such a u is not unique, we choose the smallest u according to the ordering on natural numbers. Note that $\prec_{\mathbb{D}} = \prec_{\mathbb{D}}\downarrow U$. ■

Thus, $\prec_{\mathbb{D}}$ is an ordering, under which the dependent units (those that are target of an instantaneous signal) are greater than the units that are the sources of the dependencies. The ordering is not total by this definition: one has always some freedom to arrange non-dependent units. $\prec_{\mathbb{D}}$ will be used to define the update order in the unit update cycles.

In our setting, we will do a cycle update in two steps: the first step performs all updates for the first half cycle, the second those for the second half cycle. These two steps are denoted by functions $\mathbb{T}_1^S$ and $\mathbb{T}_2^S$, working on units and delayed and instantaneous signals with

$$\mathbb{T}_{1,2}^S : \mathcal{P}(\{1,\ldots,N\}) \times [\nu] \times [\delta] \to [\nu] \times [\delta]$$

The first argument reflects the set of units that still need updating, the second is the current unit and instantaneous signal contents, the third reflects the values of the delayed signals for the next cycle. The result is a pair of new unit/asserted instantaneous signals and new delayed signals.

Thus, the function $\mathbb{T}$ on states can be defined as

$$\mathbb{T}(\mu) = \begin{cases} \perp & \text{, if } \mathcal{H}(\mu) \\ \mathcal{M}(\mathbb{T}_2^S(\{1,\ldots,N\},\nu^1,\delta^1)) & \text{, otherwise} \end{cases}$$

where the function $\mathcal{M}$ maps the half-cycle update results back to states:

$$\mathcal{M}(\nu,\delta) = (\nu \oplus \delta) \downarrow (\mathrm{Lab}^c \cup \mathrm{Lab}^\delta)$$

and ν^1, δ^1 are the results from the first half-cycle update:

$$(\nu^1,\delta^1) = \mathbb{T}_1^S(\{1,\ldots,N\},\mu,\delta_\perp)$$

The functions $\mathbb{T}_{1,2}^S$ use only updates of *units*. The update of unit u is given by functions

$$\mathbb{T}_{1,2}^u : [\nu] \to [\nu] \times [\delta]$$

mapping the contents of the unit/delayed signals/instantaneous signals to a new unit and asserted signals content and newly asserted delayed signals.

$\mathbb{T}_{1,2}^u$ are the fundamental semantics one has to give in order to describe a processor system. With them we can write $\mathbb{T}_{1,2}^S$ as

$$\mathbb{T}_i^S(U,\nu,\delta') = \begin{cases} (\nu,\delta') & \text{, if } U = \emptyset \\ \mathbb{T}_i^S(U\backslash\{u\},\nu',\delta' \oplus \delta'') & \text{, otherwise} \end{cases}$$

where
$$\begin{array}{lcl} u & = & \min_{\prec_{\mathbb{D}}}(U) \\ (\nu'', \delta'') & = & \mathbb{T}_i^u(\nu) \\ \nu' & = & \nu \oplus \nu'' \end{array}$$

$\mathbb{T}_i^S$ is well defined, as U is a finite set.

4.2.2 Abstract Components

We are primarily interested in an analysis of the finite automaton describing our system. In the DFA which will be the framework we use, we will not be able to precisely represent all components of a concrete state, for reasons of computability, efficiency, etc. E.g. the input data vector in the state will not be represented in the analysis, as we would have to represent all possible input data for all possible executions. The contents of main memory will not be represented in the analysis as well, nor the cache contents. Nonetheless, we can always find correct *approximations* for these components in the analysis. A component named n_i, for which we don't want to (or can't) have any knowledge in the analysis can always be represented by an abstract domain $\hat{D} = \{\bot, \top\}$. Here, the element $\top$ represents the complete set D_{n_i}, thus we have a concretization (cf. Section 3.2.1) $\gamma : \hat{D} \to \mathcal{P}(D_{n_i})$ with $\gamma(\top) = D_{n_i}$. On the other hand we may have an approximation for the component's values that loses some knowledge but can represent some subsets of the component's domain. The approximation for the contents of caches is such an example. Then, we have a more complex abstract domain $\hat{E}$ and a concretization $\gamma_{\hat{E}}$.

Now, modeling these components precisely would be wasted effort, as we will have either no or only limited knowledge about them in the analysis later. To make this fact explicit in the model, we introduce so called *abstract components*. These are components for whose values we will have non-exact approximations in the analysis. In contrast to abstract components, the remaining values of the components will be represented exactly in the analysis later.

E.g. the abstract caches are only able to represent what is *guaranteed* to be in the cache at a given point during execution but not everything that is in the cache. On the other hand, the contents of the prefetch queue (instruction addresses) will be represented exactly.

We introduce abstract components here already for the concrete model and semantics because this makes it easier to define the abstraction of the complete state and semantics for the analysis later. In the following discussion, abstract components will occur as abstract types (i. e. predefined meanings for D_{n_i}, not defined from atomic elements, as for the other components) and predefined functions, working on abstract types. The analysis later then only has to give abstract domains for the abstract types and abstract versions of the predefined functions.

In the remainder of this section, we will show how the unit updates $\mathbb{T}^u_{1,2}$ can be defined by giving a language and semantics for an imperative unit update. In the next two sections we will apply the techniques developed here by giving two examples for processor models.

4.2.3 A Semantics for Unit Transitions

This section will introduce an imperative language which is used to define $\mathbb{T}^u_i$. The language does three things:

- It provides a type system that is used to define the domains of the components, delayed and instantaneous signals, i. e. C_i, D_i and I_ι.
- It provides the means to define the unit update, including local variables.
- It makes explicit the dependencies on abstract components both in the type system and the program.

We will use *local variables* in our unit-update programs. These variables will be referenced by identifiers from the set Lab^l.

To keep things simple, we assume that the sets Lab^c_u, Lab^δ, Lab^ι and Lab^l are pairwise disjoint. The set Lab is the union of all these identifier sets. Additionally, we assume that the identifiers used in structure or enum declarations are disjoint from other identifiers, i. e. in declarations or signal or component names.

We will now go on by defining the semantic domains of the language. Then we will introduce the *types* of data values. With this, we show how the C_i, D_i and I_ι can be defined as the meaning of types in our language. Then the abstract syntax of the language is introduced, together with the static semantics, i. e. the side conditions on well formed programs. After that, we will give the dynamic semantics of a program and show how the $\mathbb{T}^u_i$ can be defined in its terms.

Semantic Domains

The following semantic domains are defined:

- Val the set of values
- Types the set of types
- $\mathsf{Env} = [\mathrm{Lab}^l \to \mathsf{Val}] \cup [\nu]$ the environments, containing mappings of local variables, state variables, delayed signal names and instantaneous signal names to the corresponding values. Note that $[\nu] \subseteq \mathsf{Env}$.
- Stmt the set of statements

- Expr the set of expressions

Expressions as well as signals in the language have a type. We can construct unit members and signal arguments/values from the following types. Here, the brackets $[\![]\!]$ denotes the meaning of the type. The concrete annotation of the type in the program is given first, then the abstract syntax, then the meaning.

- Int,
  Int,
  $[\![\mathsf{Int}]\!] = \{-2^{63}, \ldots, 0, \ldots, 2^{63}-1\}$ the set of 64 bit integers.

- Bool,
  Bool,
  $[\![\mathsf{Bool}]\!] = \{\mathsf{true}, \mathsf{false}\}$. The set of boolean values.

- struct t_1 $\mathrm{id}_1; \ldots;$ t_n id_n; end,
  $\mathsf{S}(\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n)$,
  $[\![\mathsf{S}(\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n)]\!] = \{f \mid f : \{\mathrm{id}_1, \ldots, \mathrm{id}_n\} \to \bigcup_{1 \le i \le n} [\![t_i]\!], f(\mathrm{id}_i) \in [\![t_i]\!]\}$.
  A record of types.

- enum $\mathrm{id}_1[(t_1)|\varepsilon]| \ldots |\mathrm{id}_n[(t_n)|\varepsilon]$ end,
  $\mathsf{E}(\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n)$,
  $[\![\mathsf{E}(\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n)]\!] = \{(\mathrm{id}_i, v_i) \mid 1 \le i \le n, v_i \in [\![t_i]\!]\}$. An enumeration with constructors. If no type is given in the syntax (case ε above), then the type t_i in the abstract syntax is the unit type $()$.

- $(t_1, \ldots, t_n)$,
  $(t_1, \ldots, t_n)$,
  $[\![(t_1, \ldots, t_n)]\!] = [\![t_1]\!] \times \cdots \times [\![t_n]\!]$. A tuple of types.

- t $\mathrm{id}[n]$,
  $[t]^n$,
  $[\![[t]^n]\!] = [\{1, \ldots, n\} \to [\![t]\!]]$. An array of values of type t with n elements.

- ,
  $()$,
  $[\![()]\!] = \{()\}$. The unit type needed for signals without arguments.

- ,
  $\mathsf{sig}(t)$,
  $[\![\mathsf{sig}(t)]\!] = [\![t]\!]$ A delayed or instantaneous signal.

- $\mathsf{abstr}(\mathrm{id})$,
  $\mathsf{abstr}(\mathrm{id})$,

$[\![\mathsf{abstr}(\mathrm{id})]\!] = P(\mathrm{id})$ An abstract type has a meaning that is given by some function P. The identifier is used to differentiate between several abstract types.

- ,
  $t_1 \times \cdots \times t_n \to t$,
  $[\![t_1 \times \cdots \times t_n \to t]\!] = [[\![t_1]\!] \times \cdots \times [\![t_n]\!] \to [\![t]\!]]$. A predefined function with n arguments of types $t_1, \ldots, t_n$ and result type t. t_i and t must not be signal types.

In addition, the meaning of all types must be finite sets. There is no concrete syntax to denote the unit type $()$ or a signal type $\mathsf{sig}(t)$. These types cannot be manipulated by programs and are only used in the semantics definitions. Also, predefined functions cannot be manipulated in the program, apart from being invoked. Furthermore, predefined functions are the only means to introduce values of abstract types in the program. This trick makes it easy to provide abstractions later by simply demanding that (correct) abstract versions of the predefined functions exist.

The set Types contains the syntax of all possible types. To denote all meanings we can represent with our type system we write $[\![\mathsf{Types}]\!]$. Thus, $[\![\mathsf{Types}]\!] = \bigcup_{t \in \mathsf{Types}} [\![t]\!]$. This way we can define the semantic domain $\mathsf{Val} = [\![\mathsf{Types}]\!]$.

For every domain C_i of a component in unit μ_u we give one type $C_i^T \in \mathsf{Types}$ and define $C_i = [\![C_i^T]\!]$. Likewise for D_i and I_ι with types D_i^T and I_ι^T.

Statements

A unit update program will simply be a statement from the domain Stmt. The statements that define Stmt are

- `Skip`: This statement does nothing
- $l \leftarrow e$, where $e \in \mathsf{Expr}$ is an expression and $l \in \mathbf{lval}$ is assigned to
- `If` e `Then` s_1 `Else` s_2 `Fi`: a conditional with condition $e \in \mathsf{Expr}$ and two statements $s_1, s_2 \in \mathsf{Stmt}$
- `Goto` *Label*, where *Label* is a program label
- `Emit` $\mathrm{id}(e)$, where $e \in \mathsf{Expr}$ is an expression and $\mathrm{id} \in \mathrm{Lab}^\delta \cup \mathrm{Lab}^\iota$ is a signal name
- s `Where` $\mathrm{id}_1 : t_1 = e_1; \ldots \mathrm{id}_n : t_n = e_n$; introduces local variables id_i in the statement $s \in \mathsf{Stmt}$. The initial value of id_i is given by the expression $e_i \in \mathsf{Expr}$. The type of variable id_i is $t_i \in \mathsf{Types}$

- `For` $\mathrm{id} = e_1$
  , *To* $e_2 : s$ introduces a loop, where the local integer variable 'id' ranges from the value of $e_1 \in \mathsf{Expr}$ to $e_2 \in \mathsf{Expr}$ and the statement $s \in \mathsf{Stmt}$ is executed for all the values assigned to id

- `Break`: This statement terminates the innermost `For` loop that it occurs in.

- $s_1\ s_2$ a sequence of statements for statements $s_1, s_2 \in \mathsf{Stmt}$

In a program, certain parts may be marked by a label. The places, where a label l may appear must not be nested in a `For` or an `If-Then-Else` or a `Where` construct. For a program $p \in \mathsf{Stmt}$, we write $p@l$ for the program from the label on to the end. A label must occur after any `Goto` statement that references it. Also, a `Break` instruction must not occur outside a `For` loop construct.

The set **lval** is defined by $l \in \mathbf{lval} \iff l \in \mathbf{name} \vee l = (\mathrm{id}_1, \ldots, \mathrm{id}_n)$. I.e. an **lval** is either a name (defined below) or a tuple of identifiers.

Expressions

The set of expressions present in the language, Expr is given by

- $p(e_1, \ldots, e_n)$ where p is a *primitive operation* or *predefined function* and $e_i \in \mathsf{Expr}$

- $(e_1, \ldots, e_n)$ a tuple of expressions $e_i \in \mathsf{Expr}$

- `Received` $\mathrm{id}(\mathrm{id}_1, \ldots, \mathrm{id}_n)$ a test on received signal $\mathrm{id} \in \mathrm{Lab}^{\delta} \cup \mathrm{Lab}^{\iota}$ and $\forall 1 \leq i \leq n : \mathrm{id}_n \in \mathrm{Lab}^{l}$.

- A name $n \in \mathbf{name}$

- A literal, i. e. either an integer n, or the values true or false or an enumeration member $\mathrm{id}(e_1, \ldots, e_n)$, $e_i \in \mathsf{Expr}$

A **name** serves either as the left hand side of an assignment or can be dereferenced to obtain a value. The set **name** contains

- id an identifier, representing a signal name, a state variable or a local variable.

- $n.\mathrm{id}$ the selection of a structure member, $n \in \mathbf{name}$

- $\mathrm{id}[e]$ an array reference, $e \in \mathsf{Expr}$

- $n.\mathrm{id}[e]$ an array reference of a struct reference, $e \in \mathsf{Expr}$, $n \in \mathbf{name}$

Note that these four rules define the same set as the rules

- id
- id$[e]$, $e \in \mathsf{Expr}$
- id$.n$, $n \in$ **name**

We will use this to perform induction on names from the front or the tail of a name.

Although there are no provisions to declare the types of the unit components and/or signals in the syntax of a program, we assume that there is a list of *type declarations* given in addition to the program. A type declaration has the form t id, where $t \in \mathsf{Types}$. Also, we will later add the convenience to add *typedefs* to the concrete syntax. A typedef introduces an abbreviation for a type, saving some space in the type declarations. In our setting, a typedef is simply expanded to its definition to give the concrete type in a declaration.

This concludes the syntax of our language. Yet, there are additional conditions on the way expressions and statements may be used. E.g. the typing of expressions must be valid, etc.

Thus, we now give the rules to determine the type of an expression. The typing of an expression $e \in \mathsf{Expr}$ is given by the following inference rules. Here $e \rhd^{\xi} t$ means that the expression e has type t under the *type environment* $\xi : \mathrm{Lab} \to \mathsf{Types}$. The initial typing environment for a program p will contain entries for the unit components, and the delayed and instantaneous signals. The function enum gives for an enum identifier id the enum type $\mathsf{E}(\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n)$ it belongs to, i. e. $\mathrm{id} = \mathrm{id}_i$ for some $1 \leq i \leq n$. The function etype gives for an enum identifier id its type in the enum, t_i.

(1) $\dfrac{}{n \rhd^{\xi} \mathsf{Int}}$, if n is an integer (2) $\dfrac{}{\mathsf{true} \rhd^{\xi} \mathsf{Bool}}$

(3) $\dfrac{}{\mathsf{false} \rhd^{\xi} \mathsf{Bool}}$ (4) $\dfrac{}{() \rhd^{\xi} ()}$

(5) $\dfrac{}{\mathrm{id} \rhd^{\xi} \xi(\mathrm{id})}$, if $\mathrm{id} \in \mathrm{dom}(\xi)$

(6) $\dfrac{}{\mathrm{id} \rhd^{\xi} \mathrm{enum}(\mathrm{id})}$, if id is an enum identifier

(7) $\dfrac{e \rhd^{\xi} \mathrm{etype}(\mathrm{id})}{\mathrm{id}\ e \rhd^{\xi} \mathrm{enum}(\mathrm{id})}$, if id is an enum identifier

(8) $\dfrac{e_1 \rhd^{\xi} [t]^n, e_2 \rhd^{\xi} \mathsf{Int}}{e_1[e_2] \rhd^{\xi} t}$ (9) $\dfrac{e \rhd^{\xi} \mathsf{S}(\mathrm{id}_1 : t_1, \ldots, \mathrm{id}_n : t_n)}{e.\mathrm{id}_i \rhd^{\xi} t_i}$

$$(10)\quad \frac{e_1 \rhd^{\xi} t_1, \ldots, e_n \rhd^{\xi} t_n}{(e_1, \ldots, e_n) \rhd^{\xi} (t_1, \ldots, t_n)}$$

$$(11)\quad \frac{\mathrm{id}_1 \rhd^{\xi} t_1, \ldots, \mathrm{id}_n \rhd^{\xi} t_n, \mathrm{id} \rhd^{\xi} \mathsf{sig}(t_1 \times \cdots \times t_n)}{\texttt{Received}\, \mathrm{id}(\mathrm{id}_1, \ldots, \mathrm{id}_n) \rhd^{\xi} \mathsf{Bool}}$$

$$(12)\quad \frac{e_1 \rhd^{\xi} t_1, \ldots, e_n \rhd^{\xi} t_n, p \rhd^{\xi} t_1 \times \cdots \times t_n \to t}{p(e_1, \ldots, e_n) \rhd^{\xi} t}, \text{ if } t, t_i \text{ contain no signal types}$$

These rules present no surprises, except for rule (12), whose condition guarantees that signal types cannot be derived by other means than coming from the typing environment. The static correctness of a program p is defined by the following inference rules. Here, $s \downarrow^{\xi} p$ where s is a statement and ξ a typing environment, as above, denotes that s is well formed in the program p under ξ.

$$(13)\quad \frac{}{\texttt{Skip} \downarrow^{\xi} p} \qquad (14)\quad \frac{}{\texttt{Break} \downarrow^{\xi} p}$$

$$(15)\quad \frac{s_1 \downarrow^{\xi} p, s_2 \downarrow^{\xi} p}{s_1\ s_2 \downarrow^{\xi} p}$$

$$(16)\quad \frac{}{\texttt{Goto}\ \mathit{Label} \downarrow^{\xi} p}, \text{ if } \mathit{Label} \text{ occurs in } p$$

$$(17)\quad \frac{e \rhd^{\xi} t, \mathrm{id} \rhd^{\xi} \mathsf{sig}(t)}{\texttt{Emit}\ \mathrm{id}\, e \downarrow^{\xi} p} \qquad (18)\quad \frac{e \rhd^{\xi} \mathsf{Bool}, s_1 \downarrow^{\xi} p, s_2 \downarrow^{\xi} p}{\texttt{If}\ e\ \texttt{Then}\ s_1\ \texttt{Else}\ s_2\ \texttt{Fi} \downarrow^{\xi} p}$$

$$(19)\quad \frac{l \rhd^{\xi} t, e \rhd^{\xi} t}{l \leftarrow e \downarrow^{\xi} p}, \text{ where } t \text{ contains no signal types}$$

$$(20)\quad \frac{e_1 \rhd^{\xi_0} t_1, \ldots, e_n \rhd^{\xi_{n-1}} t_n\, s \downarrow^{\xi_n} p}{s\ \texttt{Where}\ \mathrm{id}_1 : t_1 = e_1; \ldots \mathrm{id}_n : t_n = e_n; \downarrow^{\xi} p}, \quad \begin{array}{l} \xi_0 = \xi \\ \xi_{i+1} = \xi_i \oplus \{id_{i-1} \mapsto t_{i-1}\} \end{array}$$

$$(21)\quad \frac{e_1 \rhd^{\xi} \mathsf{Int}, e_2 \rhd^{\xi} \mathsf{Int}, s \downarrow^{\xi'} p}{\texttt{For}\ \mathrm{id} = e_1 \texttt{To}\ e_2 \downarrow^{\xi} p}, \xi' = \xi \oplus \{\mathrm{id} \mapsto \mathsf{Int}\}$$

Rule (16) does not make any assumptions on the correctness of the program fragment at $p@\mathit{Label}$, since that will be checked by the sequencing rule (15). It demands, however, that the label actually occurs in the program. The more interesting rule is rule (19): the condition makes sure that signal values can only be changed by the `Emit` construct. Also, signals in themselves cannot be used in expressions by rule (12) for expressions and the fact that they cannot be defined by a type declaration for unit components.

A program p for a unit u is statically correct iff $p \downarrow^{\xi} p$, where

$$\begin{aligned}\xi = \ & \{l \mapsto C_l^T \mid l \in \mathrm{Lab}_u^c\} \cup \\ & \{l \mapsto D_l^T \mid l \in \mathrm{Lab}^{\delta}\} \cup \\ & \{l \mapsto I_l^T \mid l \in \mathrm{Lab}^{\iota}\} \cup \\ & \{P \mapsto t_P \mid P \text{ is predefined function with type } t_P\}\end{aligned}$$

Dynamic Semantics

The dynamic semantics for a program p is described by the transformation of an environment into a modified environment together with the newly emitted delayed signals (instantaneous signals are recorded in the environment). We will give this semantics as $p \rightarrow_p^{(\rho,\delta_\perp)} (\rho', \rho^{\delta})$, meaning that the environment ρ is transformed under program p into the environment ρ' and results in delayed signals ρ^{δ} being asserted. $\delta_\perp$ is just the "inactive" environment for delayed signals. Besides the mappings for the unit components and currently active signals, the environment ρ also contains mappings for the local variables in each unit. Thus, we can write it as a mapping

$$\begin{aligned}\rho = \ & \{l \mapsto v \mid l \in \mathrm{Lab}_u^c\} \cup \\ & \{l \mapsto v \mid l \in \mathrm{Lab}^{\delta}\} \cup \\ & \{l \mapsto v \mid l \in \mathrm{Lab}^{l}\} \cup \\ & \{l \mapsto v \mid l \in L \subseteq \mathrm{Lab}^{\iota}\}\end{aligned}$$

Or, equivalently, represent it as a unit component/signal mapping extended with mappings for the local variables:

$$\rho = \nu \cup \{l \mapsto v \mid l \in \mathrm{Lab}^{l}\}$$

With this, we can give $\mathbb{T}_i^u$ in terms of the relation $\rightarrow_p$:

$$\mathbb{T}_i^u(\nu) = (\rho \downarrow (\mathrm{Lab}^c \cup \mathrm{Lab}^{\delta} \cup \mathrm{Lab}^{\iota}), \rho^{\delta}) \text{ iff } p \rightarrow_p^{(\nu,\delta_\perp)} (\rho, \rho^{\delta}) \qquad (4.2.5)$$

where p_i is the program for half-cycle update i.

We start by giving the rules for evaluating an expression under an environment ρ. Since the `Received` construct introduces side effects in expression evaluation, the result of an expression is not only a value but also a new environment. We write $e \rightarrow_{\mathcal{E}}^{\rho} (v, \rho')$ if the expression e evaluates to the value v and a new environment ρ'.

(22) $\dfrac{}{n \rightarrow_{\mathcal{E}}^{\rho} (n, \rho)}$, if n is an integer

(23) $$\frac{}{b \rightarrow^{\rho}_{\mathcal{E}} (b, \rho)}, b \in \{\mathsf{true}, \mathsf{false}\}$$

(24) $$\frac{e_1 \rightarrow^{\rho}_{\mathcal{E}} (v_1, \rho_1), \ldots, e_n \rightarrow^{\rho_{n-1}}_{\mathcal{E}} (v_n, \rho_n)}{(e_1, \ldots, e_n) \rightarrow^{\rho}_{\mathcal{E}} ((v_1, \ldots, v_n), \rho_n)}$$

(25) $$\frac{}{\mathrm{id} \rightarrow^{\rho}_{\mathcal{E}} (\rho(\mathrm{id}), \rho)}, \text{ if id} \in \mathrm{dom}(\rho)$$

(26) $$\frac{}{\mathrm{id} \rightarrow^{\rho}_{\mathcal{E}} ((\mathrm{id}, ()), \rho)}, \text{ if id is an enum identifier}$$

(27) $$\frac{e \rightarrow^{\rho}_{\mathcal{E}} (v, \rho')}{\mathrm{id}\, e \rightarrow^{\rho}_{\mathcal{E}} ((\mathrm{id}, v), \rho')}, \text{ if id is an enum identifier}$$

(28) $$\frac{e \rightarrow^{\rho}_{\mathcal{E}} (v, \rho')}{\mathrm{id}[e] \rightarrow^{\rho}_{\mathcal{E}} (\rho'(\mathrm{id})(v), \rho')}$$

(29) $$\frac{n \rightarrow^{\rho \oplus \rho(\mathrm{id})}_{\mathcal{E}} (v, \rho')}{\mathrm{id}.n \rightarrow^{\rho}_{\mathcal{E}} (v, \rho' \uparrow \mathrm{dom}(\rho(\mathrm{id})) \oplus (\rho \downarrow \mathrm{dom}(\rho(\mathrm{id}))))}$$

(30) $$\frac{}{\texttt{Received}\,\mathrm{id}(\mathrm{id}_1, \ldots, \mathrm{id}_n) \rightarrow^{\rho}_{\mathcal{E}} (\mathsf{true}, \rho \oplus \{\mathrm{id}_i \mapsto v_i \mid 1 \leq i \leq n\})}, \text{ if}$$

$$\mathrm{id} \mapsto (v_1, \ldots, v_n) \in \rho \wedge \mathrm{id} \in \mathrm{Lab}^{\iota} \text{ or}$$
$$\mathrm{id} \in \mathrm{Lab}^{\delta} \wedge \rho(\mathrm{id}) \neq \delta_{\perp}(\mathrm{id})$$

(31) $$\frac{}{\texttt{Received}\,\mathrm{id}(\mathrm{id}_1, \ldots, \mathrm{id}_n) \rightarrow^{\rho}_{\mathcal{E}} (\mathsf{false}, \rho)}, \text{ if}$$

$$\mathrm{id} \in \mathrm{Lab}^{\iota} \wedge \nexists \mathrm{id} \mapsto v \in \rho \text{ or}$$
$$\mathrm{id} \in \mathrm{Lab}^{\delta} \wedge \rho(\mathrm{id}) = \delta_{\perp}(\mathrm{id})$$

(32) $$\frac{e_1 \rightarrow^{\rho}_{\mathcal{E}} (v_1, \rho_1), \ldots, e_n \rightarrow^{\rho_{n-1}}_{\mathcal{E}} (v_n, \rho_n)}{p(e_1, \ldots, e_n) \rightarrow^{\rho}_{\mathcal{E}} ([\![p]\!](\rho_n, v_1, \ldots, v_n))}$$

These rules fix the evaluation order of expressions to be from left to right. Rules (30) and (31) are interesting. They define the value of the `Received` construct to be true if the signal with the given identifier is either an instantaneous signal and is present in the environment or is a delayed signal and is set to something different from the default value for that signal. Rule (32) uses the meaning $[\![p]\!]$ of the predefined function p, applied to the meanings of the arguments of p to define the semantics. This meaning includes the side effects that the evaluation of p has on the environment. Thus, $[\![p]\!](\rho, v) = (v', \rho')$, where v' is the value computed by p and ρ' is the new environment.

With these evaluation rules, we can then define the semantics of a program p. We write $p \rightarrow^{\tau}_{p} \tau'$ if the program p transforms the environment ρ into an environment ρ' and newly asserted delayed signals ρ^δ with $\tau' = (\rho', \rho^\delta)$. As a special case, we write $p \rightarrow_p^\tau \texttt{Break}(\tau')$ if program fragment p should be ignored after executing a `Break` instruction. In all rules, we abbreviate the pair (ρ, ρ^δ) by τ.

$$(33) \quad \frac{}{p' \rightarrow_p^{\texttt{Break}(\tau)} \texttt{Break}(\tau)}$$

$$(34) \quad \frac{}{\texttt{Break} \rightarrow_p^\tau \texttt{Break}(\tau)}\ \text{, if } \tau \neq \texttt{Break}(\tau')$$

$$(35) \quad \frac{}{\texttt{Skip} \rightarrow_p^\tau \tau}$$

$$(36) \quad \frac{e \rightarrow_{\mathcal{E}}^\rho ((v_1, \ldots, v_n), \rho')}{(\mathrm{id}_1, \ldots, \mathrm{id}_n) \leftarrow e\ \rightarrow_p^\tau (\rho' \oplus \{\mathrm{id}_i \mapsto v_i \mid 1 \le i \le n\}, \rho^\delta)}$$

$$(37) \quad \frac{e \rightarrow_{\mathcal{E}}^\rho (v, \rho')}{\mathrm{id} \leftarrow e\ \rightarrow_p^\tau (\rho'[\mathrm{id} \mapsto v], \rho^\delta)}$$

$$(38) \quad \frac{e' \rightarrow_{\mathcal{E}}^\rho (v_1, \rho_1)\ ,\ e \rightarrow_{\mathcal{E}}^{\rho_1} (v_2, \rho_2)}{\mathrm{id}[e'] \leftarrow e\ \rightarrow_p^\tau (\rho_2[\mathrm{id} \mapsto (\rho_2(\mathrm{id})[v_1 \mapsto v_2])], \rho^\delta)}\ \text{, if } 1 \le v_1 \le n$$

$$(39) \quad \frac{n \leftarrow e\ \rightarrow_p^{(\rho \oplus \rho(\mathrm{id}), \rho^\delta)} (\rho', \rho'^\delta)}{\mathrm{id}.n \leftarrow e\ \rightarrow_p^\tau (\rho' \uparrow \mathrm{dom}(\rho(\mathrm{id})) \oplus (\rho \downarrow \mathrm{dom}(\rho(\mathrm{id}))), \rho'^\delta)}$$

$$(40) \quad \frac{e \rightarrow_{\mathcal{E}}^\rho (\mathsf{true}, \rho'), s_1 \rightarrow_p^{\rho'} \rho_2}{\texttt{If}\ e\ \texttt{Then}\ s_1\ \texttt{Else}\ s_2\ \texttt{Fi} \rightarrow_p^\tau (\rho_2, \rho^\delta)}$$

$$(41) \quad \frac{e \rightarrow_{\mathcal{E}}^\rho (\mathsf{false}, \rho'), s_2 \rightarrow_p^{\rho'} \rho_2}{\texttt{If}\ e\ \texttt{Then}\ s_1\ \texttt{Else}\ s_2\ \texttt{Fi} \rightarrow_p^\tau (\rho_2, \rho^\delta)}$$

$$(42) \quad \frac{p@\mathit{Label} \rightarrow_p^\tau \tau'}{\texttt{Goto}\ \mathit{Label} \rightarrow_p^\tau \tau'} \qquad (43) \quad \frac{p@\mathit{Label} \rightarrow_p^\tau \texttt{Break}(\tau')}{\texttt{Goto}\ \mathit{Label} \rightarrow_p^\tau \texttt{Break}(\tau')}$$

$$(44) \quad \frac{e \rightarrow_{\mathcal{E}}^\rho (v, \rho')}{\texttt{Emit}\ \mathrm{id}(e) \rightarrow_p^\tau (\rho' \oplus \rho'', \rho^\delta \oplus \rho'^\delta)}\ , \quad \begin{array}{l} \rho'' = \begin{cases} \{\mathrm{id} \mapsto v\}\ \text{, if id} \in \mathrm{Lab}^\iota \\ \emptyset\ \text{, otherwise} \end{cases} \\ \rho'^\delta = \begin{cases} \{\mathrm{id} \mapsto v\}\ \text{, if id} \in \mathrm{Lab}^\delta \\ \emptyset\ \text{, otherwise} \end{cases} \end{array}$$

$$(45) \quad \frac{\begin{array}{l} e_1 \rightarrow_{\mathcal{E}}^\rho (v_1, \rho_1) \\ \ldots \\ e_n \rightarrow_{\mathcal{E}}^{\rho_{n-1}[\mathrm{id}_{n-1} \mapsto v_{n-1}]} (v_n, \rho_n)\ \ , s \rightarrow_p^{(\rho_n[\mathrm{id}_n \mapsto v_n], \rho^\delta)} \tau' \end{array}}{s\ \texttt{Where}\ \begin{array}{l} \mathrm{id}_1 = e_1; \\ \ldots \\ \mathrm{id}_n = e_n; \end{array}\ \rightarrow_p^\tau (\rho' \uparrow M \oplus (\rho \downarrow M), \rho'^\delta)}\ \text{, and } M = \{\mathrm{id}_1, \ldots, \mathrm{id}_n\}$$

$$(46)\quad \frac{s_1 \rightarrow_p^{\tau} \tau', s_2 \rightarrow_p^{\tau'} \tau_2}{s_1\ s_2 \rightarrow_p^{\tau} \tau_2}$$

$$(47)\quad \frac{e_1 \rightarrow_{\mathcal{E}}^{\rho} (v_1, \rho_1), e_2 \rightarrow_{\mathcal{E}}^{\rho_1} (v_2, \rho_2)}{\texttt{For}\ \mathrm{id} = e_1 \texttt{To}\ e_2\ s \rightarrow_p^{\tau} (\rho_2, \rho^{\delta})}\ \text{, if } v_1 > v_2$$

$$(48)\quad \frac{\begin{array}{ll} e_1 \rightarrow_{\mathcal{E}}^{\rho} (v_1, \rho_1) & s \rightarrow_p^{(\rho_2[\mathrm{id} \mapsto v_1], \rho^{\delta})} \tau_3 \\ e_2 \rightarrow_{\mathcal{E}}^{\rho_1} (v_2, \rho_2) & \texttt{For}\ \mathrm{id} = v_1 + 1 \texttt{To}\ v_2 \rightarrow_p^{\tau_3} \tau_4 \end{array}}{\texttt{For}\ \mathrm{id} = e_1 \texttt{To}\ e_2\ s \rightarrow_p^{\tau} (\rho_4 \uparrow \{\mathrm{id}\} \oplus (\rho_2 \downarrow \{\mathrm{id}\}), \rho_4^{\delta})}$$

$$(49)\quad \frac{\begin{array}{l} e_1 \rightarrow_{\mathcal{E}}^{\rho} (v_1, \rho_1) \\ e_2 \rightarrow_{\mathcal{E}}^{\rho_1} (v_2, \rho_2)\ \ s \rightarrow_p^{(\rho_2[\mathrm{id} \mapsto v_1], \rho^{\delta})} \texttt{Break}(\tau') \end{array}}{\texttt{For}\ \mathrm{id} = e_1 \texttt{To}\ e_2\ s \rightarrow_p^{\tau} (\rho' \uparrow \{\mathrm{id}\} \oplus (\rho_2 \downarrow \{\mathrm{id}\}), \rho'^{\delta})}$$

These rules use induction on the program length only, except rule (48) for the `For` construct. But here, the expressions e_1 and e_2 are evaluated only once, and because the set Int is finite, there can be only finitely many applications of those rules. Note that the constraints on the label of a `Goto` statement ensure that rule (42) performs induction on a smaller part of the program p.

Therefore, there is at most one (ρ', ρ^{δ}) for a τ with $p \rightarrow_p^{\tau} (\rho', \rho^{\delta})$, guaranteeing the well defined-ness of $\mathbb{T}_i^u$.

This concludes the concrete semantics for our state update.

For the examples in the next two sections, we assume that a set of predefined functions is available, namely addition, subtraction, multiplication, division, remainder and negation on integers. Additionally, logical negation, logical and, logical or, equality and inequality are assumed to be predefined. We will use the more usual infix or prefix notation for these operations with the corresponding symbols.

4.3 Example 1: The MCF 5307

The model for the MCF 5307 follows closely the description of its pipeline in Section 2.5 and is depicted in Figure 4.4.

The following units exist:

IAG instruction address generation

IC1 instruction fetch cycle 1

IC2 instruction fetch cycle 2

IED instruction early decode

IB instruction buffer, the first four units correspond directly to the stages of the IFP of the MCF 5307

EX execution unit. This unit summarizes the two stages of the OEP

SST store stall timer. This models a side condition on stores described in the MCF 5307 manual

BU bus unit. This models the accesses to external memory and the internal SRAM.

Figure 4.4 shows the signals between these units as arrows. In the following sections, the various units are presented in detail together with the signals they may receive and send out.

Here and in the next example we use a relaxed syntax compared to the formal syntax definition of the previous sections. Also, not every functions or expression is formulated strictly in the given syntax in order not to clutter the presentation too much.

4.3.1 Instruction Address Generation (IAG)

State and Signals

IAG carries as inner state *IAGstate* an address a, or none if the fetch pipeline has been stopped. It may receive the following signals:

1. $set_{\mathrm{EX}}(a')$ from the execution unit EX, where a' is the new address to be fetched;

2. $stop_{\mathrm{EX}}$ from the execution unit EX if the processor is halted;

3. $set_{\mathrm{IED}}(a')$ from instruction early decode IED, where a' is the new address to be fetched;

4. $stop_{\mathrm{IED}}$ from instruction early decode IED if a return instruction or computed branch is found;

5. *wait* from its successor stage IC1 if the fetch pipeline is stalled.

If more than one signal is received in the same cycle, only one of them takes effect. The signals from EX have highest priority and *wait* has lowest.

IAG may send a signal $addr(a)$ to IC1, where a is an address. Unlike the signals received by IAG, this signal is delayed, i. e.received by IC1 in the next cycle. It models the standard way of advancing the pipeline by one step.

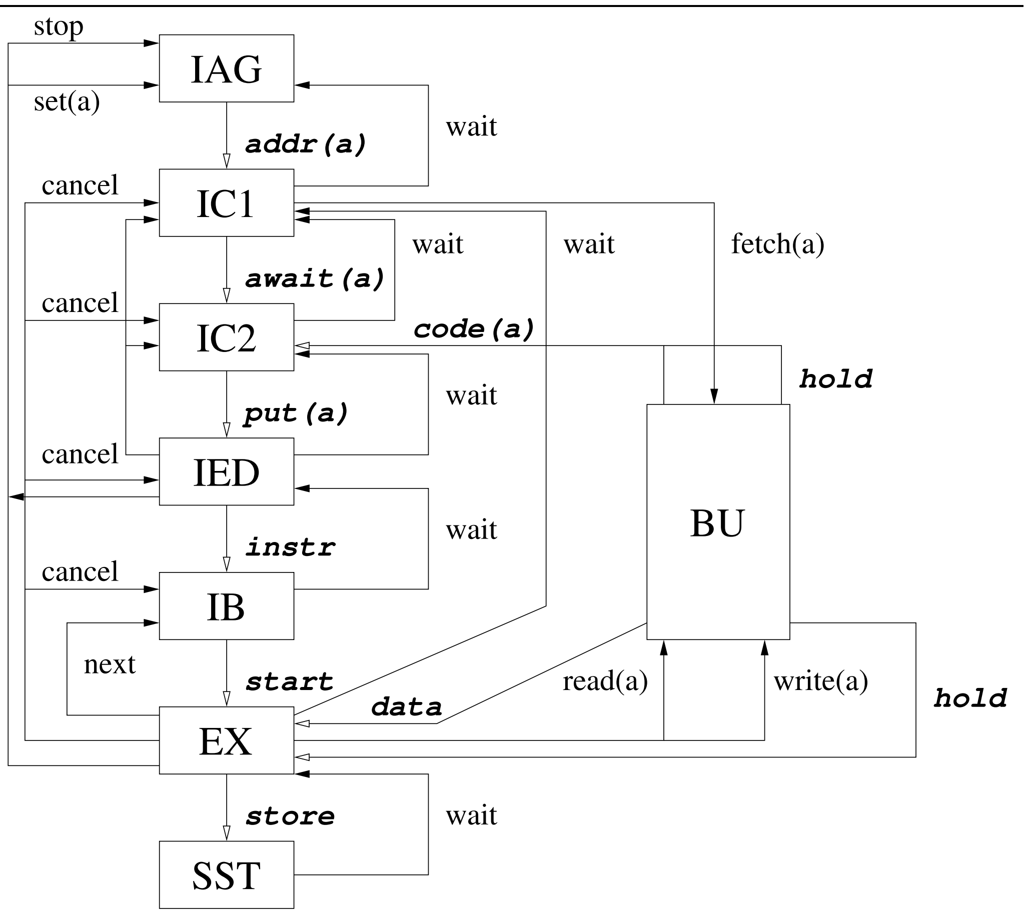


Figure 4.4: Map of formal pipeline model

State Evolution

The inner state of IAG evolves according to the following program.

```
If stop received from EX Then
    $IAGstate \leftarrow$ none
Else If set($a$) received from EX Then
    $IAGstate \leftarrow a$
    Emit addr($a$) signal
Else If stop received from IED Then
    $IAGstate \leftarrow$ none
Else If set($a$) received from IED Then
    $IAGstate \leftarrow a$
    Emit **addr**$(a)$ signal
Else If **wait** received Then
    **do nothing**
Else
    If $IAGstate \neq$ none Then
        $IAGstate \leftarrow IAGstate + 4$
        Emit **addr**$(IAGstate)$ signal
    Fi
Fi
```

4.3.2 Instruction Fetch Cycle 1 (IC1)

State and Signals

Like IAG, IC1 carries as inner state *ICIstate* an address a, or none if it is temporarily empty. It may receive the following signals:

1. *addr*(a) from IAG communicating the next address to be fetched;
2. *wait* from its successor stage IC2 if the fetch pipeline is stalled;
3. *wait* from the execution unit EX if EX needs the bus for reading or writing;
4. *cancel* from EX or IED if prefetching is redirected or stopped.

IC1 may send a *wait* signal to its predecessor IAG, a *fetch*(a) signal to the bus, where a is an address, and an *await*(a) signal to its successor IC2, preparing it to receive the code at address a some time later. The await signal is delayed.

State Evolution

The inner state of IC1 evolves in two steps:

```
If **addr**$(a)$ received Then
    $ICIstate \leftarrow a$
Fi
If **cancel** received Then
    $ICIstate \leftarrow$ none
Else If **wait** received Then
    If $ICIstate \neq$ none Then
        Emit **wait** signal
    Fi
Else
    If $ICIstate \neq$ none Then
    Emit **fetch**($ICIstate$) signal
    Emit **await**($ICIstate$) signal
    $ICIstate \leftarrow \mathsf{none}$
  Fi
Fi
```

If IC1 need not wait and contains an address a, it puts a request for the code at address a to the bus and changes its internal state to none which is usually replaced by the next address in the first step of the next cycle. If not, the setting to none prevents the same address from being fetched twice.

If IC1 is told to wait and contains none, it does not send a wait signal to IAG because it is ready to receive a new address anyway.

If IC1 must wait and already contains an address a, it sends a *wait* signal to IAG to prevent IAG from sending a new address before a is fetched.

4.3.3 Bus Unit

The cache proper can be modeled as yet another unit whose inner state is a concrete cache state.

Cache Semantics

This inner state is updated when a *fetch*(a) signal from IC1 is handled. The hardware cache will send the code at address a to IC2 some time (at least 1 cycle) after the fetch request. The exact time depends on the concrete cache state (hit or miss), but also on the status of the bus between CPU and cache, of the bus between cache and memory, of the write buffer between cache and memory, and of the memory itself. All these components must also be modeled by units with inner state, communicating by signals. The unit is left out for space and complexity reasons. All fetch signals from IC1 are answered by code signals to IC2 some time (at least 1 cycle) later.

The code itself is irrelevant in the model considered here, only the time of the answer matters. For the purpose of coordinating request and answer, we include the requested address in the answer, which is thus a signal *code*(a) to IC2.

4.3.4 Instruction Fetch Cycle 2 (IC2)

State and Signals

The inner state *ICIIstate* of IC2 is none if it is temporarily empty, $\mathsf{w}(a)$ if IC2 is waiting for the code at address a, and $\mathsf{r}(a)$ if it has received the code.

IC2 may receive the following signals:

1. *await*(a) from IC1 communicating the address that is currently fetched;
2. *code*(a) from the bus controller if the code is sent now;
3. *wait* from its successor unit Instruction Early Decode (IED) if the byte buffer in IED is full;
4. *cancel* from EX or IED if prefetching is redirected or stopped.

IC2 may send a *wait* signal to its predecessor IC1, or a delayed *put*(a) signal to its successor IED, where a is an address.

State Evolution

The inner state of IC2 evolves in three steps:

```
If $\textbf{await}(a)$ received Then
  $\mathit{ICIIstate} \leftarrow \mathsf{w}(a)$
Fi
If $\mathsf{w}(a') == \mathit{ICIIstate} \wedge \textbf{code}(a')$ received Then
  $\mathit{ICIIstate} \leftarrow \mathsf{r}(a')$
Fi
If **cancel** received Then
  $\mathit{ICIIstate} \leftarrow$ none
Else If **wait** received Then
  Emit **wait** signal
Else
  If $\mathit{ICIIstate} = \mathsf{r}(a'')$ Then
    $\mathit{ICIIstate} \leftarrow$ none
    Emit $\textbf{put}(a'')$ signal
  Else If $\mathit{ICIIstate} = \mathsf{w}(a'')$ Then
    Emit **wait** signal
  Fi
Fi
```

4.3.5 Instruction Early Decode (IED)

State and Signals

The inner state *IEDstate* of IED is a pair (b,q), where b is the number of bytes in the byte buffer of the IED, and q is a queue of instructions. The number b is even and satisfies $0 \le b \le 8$, under the assumption that 8 is the maximal capacity of the byte buffer. The queue $q = [i_n, \ldots, i_1]$ contains all instructions which start in the byte buffer; the last one (i_n) may be incomplete. The maximal length of q is $8/2 = 4$.

IED may receive the following signals:

1. *put*(a) from IC2 if new code is arriving;
2. *wait* from its successor unit, the Instruction Buffer (IB), if the buffer is full;
3. *cancel* from the execution unit EX if prefetching is redirected.

IED may send the following signals:

1. *wait* to its predecessor IC2 if the byte buffer is full;
2. $set_{\mathrm{IED}}(a)$ to the Instruction Address Generation IAG if the fetch pipeline is redirected to address a;
3. $stop_{\mathrm{IED}}$ to the Instruction Address Generation IAG if the fetch pipeline is stopped;
4. *cancel* to IC1 and IC2 if the fetch pipeline is redirected or stopped;
5. *instr*, a delayed signal sent to IB when an instruction is forwarded to IB (the instruction itself is irrelevant for the purpose of this model).

State Evolution

Step 1 reacts to *put*(a) signals

```
If put(a) received Then
  $IEDstate \leftarrow (b+4, I_{a+2}.I_a.q)$
  where $I_a$ is the list of instructions starting at address $a$, which is either
   a singleton or empty.
  $I_{a+2}$ is defined analogously, and '.' is list concatenation
  and $(b,q) = IEDstate$
Fi
```

Step 2:

```
If cancel received from EX Then
  $mvIEDstate \leftarrow (0, [\,])$
Fi
```

Step 3

```
If **wait** not received Then
  If $IEDstate$ contains complete instruction $i$
  i. e.$q = q'.[i]$ and $b \geq w = width(i)$ where
  $IEDstate = (b,q)$ Then
```

```
    Step 3a: jump, call decode
    If $i$ is an unconditional jump or call with target $a$
    or a conditional jump with target $a$ which is predicted as taken
    Then
      Emit $set_{\mathrm{IED}}(a)$ signal
      Emit $cancel$ signal
      $IEDstate \leftarrow (w, [i])$
    Fi
    Step 3b:
    If $i$ is a return instruction or indirect jump Then
      Emit $stop_{\mathrm{IED}}$ signal
      Emit $cancel$ signal
      $IEDstate \leftarrow (w, [i])$
    Fi
  Step 4
  Emit $instr$ signal
  $IEDstate \leftarrow (b - w, q')$ where $IEDstate = (b, q'.[i])$ and
  $w = width(i)$
Fi
Step 5
If $IEDstate = (b, q)$ with $b + 4 > 8$ Then
  Emit $wait$ signal
Fi
```

4.3.6 Instruction Buffer (IB)

State and Signals

The instruction buffer can hold up to 8 instructions. Its purpose is the separation of the fetch pipeline (IAG till IED) from the execution pipeline (EX, see below).

The actual instructions in the buffer are irrelevant for the purpose of this model; only the number of instructions matters. Thus, the inner state *IBstate* of IB is an integer i with $0 \leq i \leq 8$.

IB may receive an *instr* signal from IED when a new instruction is arriving, a *cancel* signal from the execution unit EX when fetching is stopped or redirected, and a *next* signal from EX indicating that EX is ready to start the execution of the next instruction. The *next* signal is repeated if IB is not able to forward the next instruction.

IB may send a *wait* signal to IED when the instruction buffer is full, and a *start* signal to EX when a new instruction is ready to start. The *start* signal is only sent in response to a *next* signal. It is delayed.

State Evolution

The inner state of IB evolves in four steps.

```
Step 1:
If cancel received Then
    $IBstate \leftarrow 0$
Else
    Step 2:
    If instr received Then
        $mvIBstate \leftarrow IBstate + 1$
    Fi
    Step 3:
    If next received and $IBstate > 0$ Then
        $IBstate \leftarrow IBstate - 1$
        Emit start signal
    Fi
    Step 4:
    If $IBstate = 8$ Then
        Emit wait
    Fi
Fi
```

With this modeling, an instruction can enter and leave an empty instruction buffer in the same cycle (in the ColdFire manual [Mot00], it is indicated that an empty instruction buffer can be bypassed). Yet a full instruction buffer refuses to accept another instruction in the next cycle even if an instruction can be forwarded to the execution unit in that cycle because this is not known in advance.

4.3.7 Store Stall Timer (SST)

Before we come to the description of the execution unit EX, we describe SST, an auxiliary unit that implements a sequence-related pipeline stall described as follows in the ColdFire manual [Mot00]:

> This type of stall involves consecutive store operations, excluding the MOVEM instruction. For all store operations (except MOVEM), certain hardware resources within the processor are marked as “busy” for two clock cycles after the final DSOC cycle of the store instruction. If a subsequent store instruction is encountered within this two-cycle window, it is stalled until the resource again becomes available.

State and Signals

Hence, the inner state *SSTstate* of SST is an integer t with $0 \leq t \leq 2$. SST is activated by a delayed *store* signal from EX and may send *wait* signals to EX, which are ignored unless another store operation is performed.

State Evolution

SST evolves in two steps:
Step 1:

```
If received store Then
    $SSTstate \leftarrow 2$
Fi
```

Step 2:

```
If $SSTstate > 0$ Then
    $SSTstate \leftarrow SSTstate - 1$
    Emit wait signal
Fi
```

4.3.8 Execution Unit (EX)

State and Signals

EX is the most complex unit of this pipeline model. It may receive the following signals:

1. *start* (delayed) from IB when a new instruction is ready to be executed.
2. *wait* from SST indicating that a store operation has been performed recently.
3. *data* from the bus controller indicating that data is being sent to EX. The actual data values are irrelevant for the purpose of this model.

EX may send the following signals:

1. *next* to IB when EX is ready to execute the next instruction.
2. *store* (delayed) to SST to prevent store operations in the next two cycles.
3. $read(a)$ to the bus controller when EX wishes to read the memory at address a. (For the moment, we assume that this address is statically known.) This request will be answered by a *data* signal in the next cycle or some time later. EX is stalled until the answer arrives.

4. *write*(a) to the bus controller when EX wishes to write to the memory at address a. (For the moment, we assume that this address is statically known.) The actual data value that is written is irrelevant in this model. There is no answer, so EX need not wait (except for SST induced stalls).

5. *wait* to IC1 when EX needs the bus connection for data accesses. This signal prevents IC1 from using the bus for fetching code.

6. $\mathit{stop}_{\mathrm{EX}}$ to IAG when the processor is halted.

7. $\mathit{set}_{\mathrm{EX}}(a)$ to IAG when prefetching should continue at a.

8. *cancel* to IC1, IC2, IED, and IB when the processor is halted or prefetching is redirected.

The behavior of EX is derived from schedules telling what each instruction does in each cycle. The schedule for an instruction heavily depends on the structure of the operands of the instruction; memory operands need to access memory while register operands need not. To be more precise, schedules are associated with the edges of the control flow graph. For instance, the two edges starting from a conditional branch instruction (one to the textual successor, one to the branch target) carry two different schedules since only one of them includes a redirection of the fetch pipeline.

The inner state of EX is a schedule (a suffix of a schedule associated with an edge). A *schedule* is a list of items. An *item* is either a set of *events* to be performed within one cycle, or a *control item* that conceptually needs no time. The following events exist:

- read(a) telling that the instruction reads from memory address a in this cycle (more exactly, it puts the request for reading a to the bus).
- write(a) telling that the instruction writes to memory address a in this cycle.
- stop when the processor is halted.
- fetch(a) when prefetching is redirected to address a.

The following control items exist:

- await indicating that EX is awaiting a *data* signal.
- stall indicating that the next event set contains a store operation that may be stalled by SST.
- store indicating that the previous event set contained a store operation that induces SST stalling.

await items are inserted dynamically into schedules when running them, but stall and store control items must already be present in the schedules associated with control flow edges. The reason is that not all store operations are followed by store items (the ones in MOVEM not).

A set of events may be empty. We assume that the schedule for every edge starts with an empty event set, i. e. instructions do not perform anything relevant in their first cycle. This is important since the first cycle of an instruction may run in parallel with the last cycle of the previous one. This first cycle is in fact not modeled explicitly, but implicitly by the fact that the *start* signal from IB to EX is delayed.

State Evolution

The evolution of EX is modeled by the following six steps.

Step 1:
```
If start received Then
  $EXstate \leftarrow sched(acti)$
  where sched gives the schedule for an instruction
  and acti is the instruction being analyzed currently
Fi
```
Step 2:
```
If $EXstate = \mathsf{await}.s$ Then
  If data received Then
    $EXstate \leftarrow s$
  Fi
Fi
```
Step 3:
```
If $EXstate = \mathsf{stall}.s$ Then
  If wait not received Then
    $EXstate \leftarrow s$
  Fi
Fi
```
Step 4:
```
If $EXstate = S.s$ ($S$ an event set) Then
  If $\mathsf{write}(a) \in S$ Then
    Emit $write(a)$ signal
    Emit wait signal to IC1
  Fi
  If $\mathsf{read}(a) \in S$ Then
    Emit $read(a)$ signal
    Emit wait signal to IC1
    Fi
    If $\mathsf{stop} \in S$
        Emit $stop_{\mathrm{EX}}$ signal to IAG
        Emit $\mathit{cancel}$ signals to IC1, IC2, IED, and IB
    Fi
    If $\mathsf{fetch}(a) \in S$ Then
        Emit $set_{\mathrm{EX}}(a)$ signal to IAG
        Emit $\mathit{cancel}$ signals to IC1, IC2, IED, and IB
    Fi
    If $\mathsf{read} \in S$ Then
        $EXstate \leftarrow \mathsf{await}.s$
    Else
        $EXstate \leftarrow s$
    Fi
Fi
Step 5:
If $EXstate = \mathsf{store}.s$ Then
    Emit $\mathit{store}$ signal
    $EXstate \leftarrow s$
Fi
Step 4:
If $EXstate = []$ Then
    Emit $\mathit{next}$ signal
Fi
```

4.3.9 State Predicates

The four state predicates $\mathcal{F}$, $\mathcal{H}$, $\mathcal{N}$ and $\mathcal{R}$ from Definition 4.1.3 remain to be defined for this model.

For this we introduce three new state components *mcfRetired*, *mcfNext* and *mcfHalted*:

- *mcfRetired* is a Boolean that is set in Step 4 of the EX update. So we add the assignment $mcfRetired \leftarrow \mathsf{true}$ there.

- *mcfNext* is the address of the next instruction. When we finish the schedule for an instruction in step 4 of the EX update, we add the assignment $mcfNext \leftarrow target(acti)$ there. Here, *target* is the target of the instruction, which is the next address after the instruction for non branching instructions and the target of the branches otherwise.

- *mcfHalted* signals that the last instruction of the program has been retired. This is set to true in step 4 of the EX update also in that case.

Then the predicates can be defined by

- $\mathcal{F}(n,s) := s(\mathit{mcfRetired})$
- $\mathcal{N}(n,s) := n = s(\mathit{mcfNext})$
- $\mathcal{H}(s) := s(\mathit{mcfHalted})$
- $\mathcal{R}(n,s) := s[\mathit{mcfRetired} \leftarrow \mathsf{false}]$

4.4 Example 2: The PPC 755

A model for the PPC750 can be given by modeling the functional units and introducing signals exchanged between them. The evolution of the pipeline is then modeled cycle-wise by giving an update order of the functional units and describing how their inner state evolves depending on the state and signals received. Signals come in two flavors: instantaneous and delayed. Instantaneous signals are received in the same cycle as they are generated, while delayed signals are generated in one cycle and received in the next cycle.

We suggest the following functional units in the pipeline model, cf. Figure 4.5:

- **CSU**: the chip set unit. It receives signals from the **BU** requesting transfers. It delivers the data to the **BU** via signals. This unit models the memory and other devices in the system, e.g. SDRAM and the PCI bus.
- **BU**: the bus unit. It receives signals requesting instruction fetch and/or data reads/writes. It emits answering signals to these requests. The instruction/data caches are modeled in this unit.
- **FBPU**: the fetch and branch prediction unit. This unit is the most complex unit in the model, since it has to reflect the complicated dependencies between instruction fetching, dispatching, branch folding and fall-through, coupled with speculative execution. It requests instructions from the **BU** and emits signals to reflect possible dispatch of instructions from the IQ as well as mispredictions of speculative branches and correct predictions. It receives instructions from the **BU** and dispatch notifications from the **DU**.
- **DU**: the dispatch unit models the dispatch rules of the instructions in IQ0/1, which it receives from the **FBPU**. It dispatches instructions according to their resource needs and the status of the functional units, which it monitors

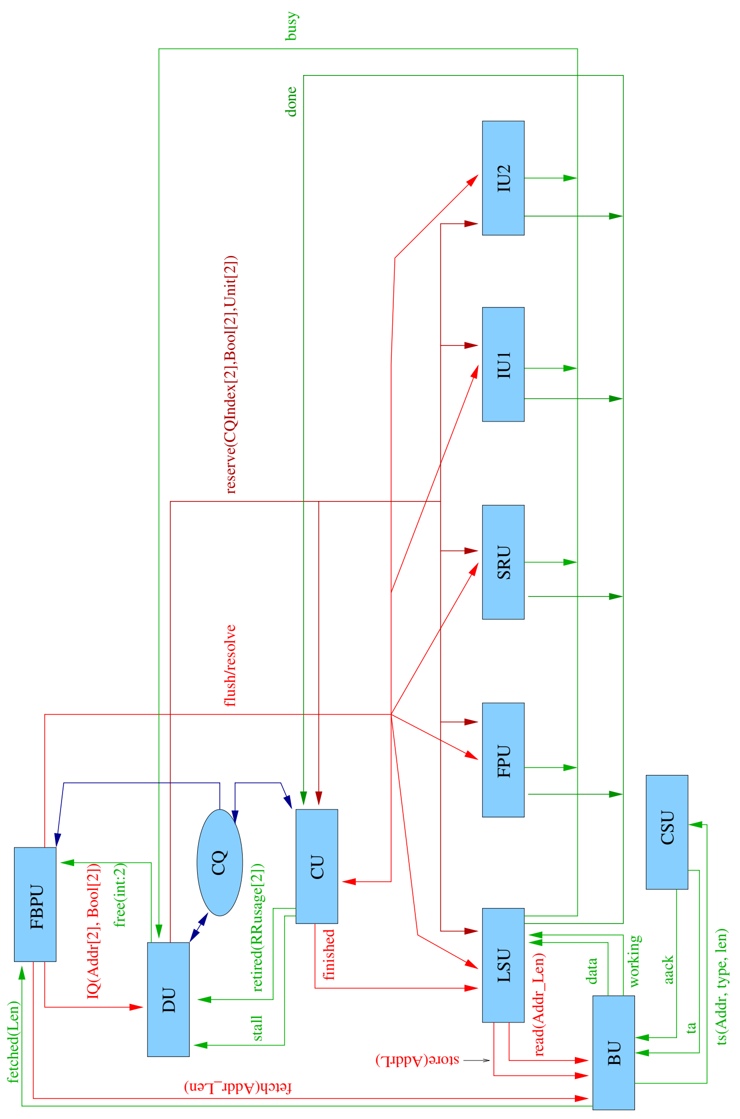


Figure 4.5: Model of the pipeline

from signals sent to it. Here also the execution, completion and refetch serialization instruction semantics is handled by delaying dispatch until other instructions have completed.

- **IU1** and **IU2**: the integer units directly model the concrete units. They receive instructions from the **DU** and emit signals if they are busy or have completed an instruction.

- **SRU**: the system register unit is modeled in a similar way to the **IU1/2** units.
- **LSU**: the load store unit. This unit is a little bit more complex, since it is pipelined and has to reflect this inner pipeline. In addition, some instructions (e.g. cache manipulation instructions) must be handled in a special way. It emits read requests to the **BU**, and store requests for multiple store instructions. Single stores are handled by the **CU**. It receives read acknowledgements from the **BU** and keeps track of stores and a busy **BU** to make sure that the internal data bus is single threaded.
- **FPU**: the floating point unit is modeled to reflect its three stages and the fact that special instructions occupy the whole pipeline.
- **CU**: the completion unit sends out retirement signals and holds to prevent the **DU** from issuing instructions. It receives instructions from the **DU** and the completion acknowledgments from the functional units. It signals the **LSU** when a store instruction retires.
- **CQ**: the completion queue is a common data pool, containing the dispatched instructions and their status.

The update order of the units is: **FBPU**, **DU**, **CU**, **LSU**, **FPU**, **IU1**, **IU2**, **SRU**, **BU** and **CSU**.

The following signals are sent and received by/from these units, as indicated in Figure 4.5:

- Instantaneous signals:
  - **IQ(Addr[2],Bool[2])**: from **FBPU** to **DU**. This signal contains the addresses of the two instructions in IQ0/1 (or none if a slot is empty), together with flags indicating if these instructions are executed speculatively.
  - **flush**: from **FBPU** to **CU** and the functional units (FUs). This indicates that a speculative branch was mispredicted and all speculative instructions should be discarded.
  - **resolve**: from **FBPU** to **CU** and the FUs. This signal indicates that a speculation has been resolved correctly. All speculative instructions now become non-speculative.

  - **reserve(CQIndex[2],Bool[2],Unit[2])**: from **DU** to **CU** and the FUs. This signal denotes the dispatch of the instructions with the corresponding indices in CQ (or none if a slot is empty) to the functional unit indicated by the third argument. The second argument is again a flag denoting speculative execution of this instruction.
  - **fetch(AddrLen)**: from **FBPU** to **BU**. This signal requests instructions to be fetched by the bus unit. The address and size of the instruction block to fetch is given by the argument.
  - **read(AddrLen)**: from **LSU** to **BU**. This signal requests a data read of the given address and length to be performed.
  - **store(AddrLen)**: from **LSU** to **BU**. This signal requests a data store of the given address and length to be performed.
  - **finished**: from **CU** to **LSU**. This signal flags the retirement of a single-store operation.

- The delayed signals:
  - **fetched(int:3)**: from **BU** to **FBPU**. This signals the completion of an instruction fetch. The length of the fetched instruction block in words is given as argument to account for cases where less instructions than requested could be fetched[6].
  - **free(int:2)**: from **DU** to **FBPU**. This gives the number of instructions to remove from the beginning of the IQ in the next cycle due to dispatch in this cycle.
  - **stall**: from **CU** to **DU**. This signal inhibits dispatch of new instructions to implement serialization semantics.
  - **busy**: from each FU to **DU**. A functional unit sending this signal cannot accept another instruction being dispatched to it.
  - **retired(RRusage[2])**: from **CU** to **DU**. This signals the retirement of up to two instructions. It gives the amount of rename registers freed by the retirement of the instructions (none, if a slot is unused).
  - **done**: from each FU to **CU**. This indicates that the first instruction in the functional unit has completed execution.
  - **data**: from **BU** to **LSU**. Indicates that a read request has finished and the data is available.
  - **working**: from **BU** to **LSU**. This signal indicates that the bus unit cannot accept another read or write request in the next cycle.

[6]It is unclear if this behavior actually happens.

- **TS**(a,t,l): from **BU** to **CSU**. This signal indicates the start of an access at address a of type $t \in \{D,S,I\}$ for l double words. The type corresponds to data reads, data stores and instruction fetches. If $l = 0$ then this indicates a sub double word access.
- **AACK**: from **CSU** to **BU**. This signals the acknowledge of the **TS** signal by the **CSU**.
- **TA**: from **CSU** to **BU**. This signal is the indication that the next data beat of a transfer has finished.

In the following sections the units will be discussed in more detail, including their inner state and the cycle evolution.

When giving the inner state we will use a C-like declaration style, using arrays with square brackets denoting element access and structures with dot (.) denoting component selection.

An important concept is that of the *index*. An index denotes an element in the IQ or the CQ. An index can be of type IQIndex if it points into the IQ, of type CQIndex if it points into the CQ or of type Index if it can point into either one. Since the IQ and CQ are modified very often, all indices have to be adjusted if elements are removed from the queues. In the following when we say that "indices are updated" we mean that all indices in all states are automatically adjusted.

4.4.1 FBPU

Inner State

The following types are defined:

Name	Component	Description
IQ	Addr address	Address of instruction
	int predLevel	Prediction level of instruction
State	WAIT(Addr, len)|	waiting for instructions to arrive from bus
	STOP(Index, Addr) |	stopped
	IGNR(Addr, l) |	will ignore next instructions arriving from bus
	RUN(Addr) |	Unit is running normally
	HOLD(Addr)	Unit has encountered too many predictions

This means that IQ is a record (struct) with two components, namely address of type Addr and predLevel of type int. In contrast, State is a kind of union of 5 alternatives, where each alternative is marked by a tag (e.g. WAIT) and carries some data (e.g. something of type Addr). The address in the state is in most cases the address of the next instruction to be fetched. In the WAIT case, it is the address of the first instruction to arrive from the bus.

The inner state of FBPU is the following record (struct):

Type	Name	Description
int:3	predLevel	Global State of execution prediction
IQ	iq[6]	IQ entries
State	state	Global unit state
IQIndex	insIndex	Index into iq of instruction currently considered
Index	CRDep[2]	CR dependencies for first two predictions
Addr	altAddr[2]	Alternative address for prediction

Here, *iq* contains the instructions in the IQ. An entry contains the address of the instruction and its prediction level. This level is 0, if the instruction is known to be executed, 1 if it follows a speculative branch, 2 if it follows two speculative branches. If the address of an entry in *iq* is none, then this entry is empty. Since the IQ is a FIFO, if an entry $iq[n]$ is empty, so are all entries $iq[n+1],\dots,iq[5]$. We use $last(iq)$ to give the index of the last (i.e. with the highest index) entry with an address not equal to none. If the IQ is completely empty, then $last(iq) = -1$. By $free(iq)$ we denote the number of entries that are empty.

$last_{CR}(i)$ is the index of the last instruction that writes into CR and comes before the instruction with index i; $last_{CTR}(i)$ is defined likewise.

State Evolution

The inner state of the FBPU evolves in one cycle by executing the following micro steps in the given order:

Step 1: This step throws out entries in IQ which have been dispatched in the cycle before

If a **free**(n) signal is received, then the entries $iq[0],\dots,iq[n-1]$ are discarded i.e. $iq[0] \leftarrow iq[n],\dots$ etc. The last n entries in *iq* that are not empty are set to be empty.
insIndex is adjusted.

Step 2: This step starts the fetching after a STOP condition has resolved

If *state* is STOP(*idx*,*addr*) and instruction denoted by *idx* has completed in CQ, then *state* ←RUN(*addr*)

Step 3: This step inserts fetched instructions into the IQ

```
If state is WAIT(addr, l) and fetched(len) has been received, then
    If $len = l$ then
        $state \leftarrow$ RUN($addr + 4 * len$)
    Else
        $state \leftarrow$WAIT($addr + 4 * len, l - len$)
    Fi
Append $len$ instructions at the end of $iq$,
i.e.  $i = last(iq)$;
      $iq[i+1] \leftarrow (addr, predLevel)$;
      ...
      $iq[i+len] \leftarrow (addr + 4 * (len-1), predLevel)$
Here $predLevel$ comes from the state. $insIndex$ is set to the index in $iq$ of the
first new instruction, i.e. $insIndex \leftarrow i+1$.
```

Step 4: This step skips over ignored instruction fetches

```
If $state$ is IGNR($addr, l$) and **fetched**($x$) is received then
    If $x = l$ then
        $state \leftarrow$RUN($addr$)
    Else
        $state \leftarrow$IGNR($addr, l - x$)
    Fi
Fi
```

Step 5: This step checks for resolved predictions

Step 5a: Check for resolving of first level prediction

```
If $d_1 = CRDep[0] \neq \mathsf{none}$ and $d_1$ has completed in CQ then
    **CR dependency resolved**
    $CRDep[0] \leftarrow \mathsf{none}$
    $good \leftarrow$ 'prediction was correct'
    **Dependency resolved: decrement prediction level, restart fetching**
    $predLevel \leftarrow predLevel - 1$
    If $good$ then
        **Our prediction was right. Just adjust instructions in IQ**
        If $state$=HOLD($a$) then
            $state \leftarrow$ RUN($a$)
            $iq[insIndex].pred \leftarrow iq[insIndex].pred - 1$
        Fi
        $\forall 0 \leq i \leq last(iq) : iq[i].pred \leftarrow max(0, iq[i].pred - 1)$
        emit **resolve** signal
    Else
        **Prediction was wrong: flush instructions, redirect**
        $\forall 0 \leq i \leq last(iq)$ :
            If $iq[i].pred > 0$ then
            $iq[i] \leftarrow (\mathsf{none}, 0)$
          Fi
        emit **flush** signal
        $predLevel \leftarrow 0$
        $insIndex \leftarrow \mathsf{none}$
        clear $CRDep$
        $state \leftarrow \begin{cases} \mathrm{IGNR}(altAddr[0]) & \text{, if } state = \mathrm{WAIT}(x) \text{ or } \mathrm{IGNR}(y) \\ \mathrm{RUN}(altAddr[0]) & \text{, otherwise} \end{cases}$
      Fi
      **Move second dependency to first (if any), clear second**
      move $CRDep[1]$, $altAddr[1]$ into $CRDep[0]$, $altAddr[0]$
      clear $CRDep[1]$, $altAddr[1]$
      **Look at second dependency, which is the first now**
      $ind \leftarrow 0$
    Else
      **Look at second dependency**
      $ind \leftarrow 1$
    Fi
```

Step 5b: Maybe resolve second level prediction

```
If $d_1 = CRDep[ind] \neq \mathsf{none}$ and $d_1$ has completed in CQ then
  **CR dependency resolved**
  $CRDep[ind] \leftarrow \mathsf{none}$
  $good \leftarrow$ 'prediction was correct'
  **Dependency resolved: decrement prediction level, restart fetching**
  $predLevel \leftarrow predLevel - 1$
  Clear $CRDep[ind]$, $altAddr[ind]$
  If $good$ then
    If $state$=HOLD($a$) then
      **If holded, reset prediction level of holding branch**
      $state \leftarrow \mathrm{RUN}(a)$
      $iq[insIndex].pred \leftarrow iq[insIndex].pred - 1$
    Fi
    **Our prediction was right. Just adjust instructions in IQ**
    $\forall 0 \leq i \leq last(iq)$ :
      If $iq[i].pred > ind)$ then
        $iq[i].pred \leftarrow max(0, iq[i].pred - 1)$
      Fi
  Else
    **Prediction was wrong: flush instructions, redirect**
```

```
        $\forall 0 \le i \le last(iq)$ :
            If $iq[i].pred > ind$ then
                $iq[i] \leftarrow (\mathsf{none}, 0)$
            Fi
        $insIndex \leftarrow \mathsf{none}$
        $state \leftarrow \begin{cases} \text{IGNR}(altAddr[ind]) & \text{, if } state = \text{WAIT}(x) \text{ or IGNR}(y) \\ \text{RUN}(altAddr[ind]) & \text{, otherwise} \end{cases}$
    Fi
Fi
```

Step 6: This step handles decoding of instructions and branch folding, etc

```
If $insIndex > last(iq)$ or $insIndex = \mathsf{none}$ then
    $insIndex \leftarrow \mathsf{none}$
    Goto Step 7
Fi
Let $i$ be the instruction at address $iq[insIndex].addr$
```

Step 6a: If instruction uses CTR or LR and CTR/LR is already written: Stop

```
If $uses(i) \cap \{CTR, LR\} \neq \emptyset$ and
    $\exists i' \in CQ : i'$ not completed and
    $write(i') \cap uses(i) \neq \emptyset$ or
    $\exists i' \in iq[0..insIndex - 1] : write(i') \cap uses(i) \neq \emptyset$ then
    $state \leftarrow \text{STOP}(li, addr+4)$
    where $li$ is the highest index of the instruction $i'$ that
        fulfills the condition in $cq[0..5]iq[0..insIndex - 1]$ and $(addr, pl) = last(iq)$.
    Goto Step 7
Fi
```

Step 6b: This step handles refetch serialization

```
If $i$ is a refetch serialization instruction then
    Clear $iq[insIndex + 1], \ldots iq[last(iq)]$
    $state \leftarrow \text{STOP}(insIndex, last(iq).addr + 4)$
Fi
```

Step 6c: This step skips non branches

```
If $i$ is not a branch then goto Step 6f
```

Step 6d: This step handles resolved branches

```
If the branch condition of $i$ is resolved then
    If $i$ is taken then
    Taken branch
    $iq[j] \leftarrow (\mathsf{none}, 0)$
    $\ldots$
    $iq[last(iq)] \leftarrow (\mathsf{none}, 0)$
    where $j = \begin{cases} insIndex + 1 & \text{, if } i \text{ updates the LR or CTR} \\ insIndex & \text{, otherwise} \end{cases}$
    $state \leftarrow$RUN($addr$), where $addr$ is the target address of the branch
    $insIndex \leftarrow \mathsf{none}$
    Goto Step 7
  Else
    Fall through branch
    Goto Step 6f
  Fi
Fi
```

Step 6e: This step handles predicted branches

```
$predLevel \leftarrow predLevel + 1$
$\forall insIndex \leq j < last(iq)$ :
  $iq[j].pred \leftarrow predLevel$
If $predLevel \geq 3$ then
  Hold because of too many predictions
  $state \leftarrow$ HOLD($last(iq).addr + 4$)
  Goto Step 7
Fi
If $predLevel < 3$ then
  Fold or fall-through based on prediction
  Let $t$ and $a$ be the predicted target and alternative target of $i$:
    $altAddr[predLevel - 1] \leftarrow a$
    $CRDep[predLevel - 1] \leftarrow \begin{cases} \mathsf{none} & \text{, if } i \text{ does not depend on CR} \\ last_{CR}(i) & \text{, otherwise} \end{cases}$
    If $i$ predicted taken then
      Fold away following instructions
      If $i$ updates LR or CTR then
        $j \leftarrow insIndex + 1$
      Else
        $j \leftarrow insIndex$
      Fi
      $iq[j] \leftarrow (\mathsf{none}, 0)$
      $\ldots$
      $iq[last(iq)] \leftarrow (\mathsf{none}, 0)$
      $state \leftarrow \begin{cases} \mathrm{IGNR}(t) & \text{, if } state = \mathrm{WAIT}(x) \\ \mathrm{RUN}(t) & \text{, otherwise} \end{cases}$
      $insIndex \leftarrow$ none
      Goto Step 7
    Fi
Fi
```

Step 6f: Look at next instruction

```
$insIndex \leftarrow insIndex + 1$
Goto Step 6
```

Step 7: This step issues next fetch

```
If $free(iq) \neq 0$ and $state$ is RUN($addr$) then
  Emit **fetch**(($addr, len$)) signal
  $state \leftarrow$ WAIT($addr$)
  where $len = min(free(iq), 4)$.
```

Step 8: This step signals dispatchable instructions

```
$a[0] \leftarrow a[1] \leftarrow$ none
$p[0] \leftarrow p[1] \leftarrow$ false
$s \leftarrow 0$
If $iq[0].addr \neq$ none then
  If instruction at $iq[0].addr$ is a fall through instruction then
    **Remove fall through branch**
    $iq[0] \leftarrow iq[1]$
    ...
    $iq[last(iq) - 1] \leftarrow iq[last(iq)]$
    Adjust $insIndex$
  Else
    $s \leftarrow s + 1$
    If $iq[0].pred < 2$ then
      **Not twice predicted instruction**
  $a[0] \leftarrow iq[0].addr$
    $p[0] \leftarrow (iq[0].pred > 0)$
    Fi
  Fi
Fi
If $iq[s].addr \neq$ none then
  **Second entry in IQ**
  If instruction at $iq[s].addr$ is a fall through instruction then
    **Remove fall through branch**
    $iq[s] \leftarrow iq[s+1]$
    $\ldots$
    $iq[last(iq)-1] \leftarrow iq[last(iq)]$
    Adjust $insIndex$
  Else
    If $iq[s].pred < 2$ then
      **Not twice predicted instruction**
      $a[s] \leftarrow iq[s].addr$
      $p[s] \leftarrow (iq[s].pred > 0)$
    Fi
  Fi
Fi
If $s > 0$ then
  Emit **IQ**$(a,p)$ signal
Fi
```

4.4.2 CQ

The **CQ** only contains state that is accessed by **FBPU**.

The following types are defined:

Name	Component	Description
Unit	Branch |	Pseudo unit for branches
	IU1 |	
	IU2 |	
	SRU |	
	LSU |	
	FPU |	
	none	No unit assigned
CQ	Addr addr	Address of instruction
	Bool spec	Is Executed speculatively
	Bool complete	Is finished
	Bool ready	Is ready to execute
	Unit unit	Unit this instruction is executed on

The **CQ** contains the following state:

Type	Name	Description
CQ	cq[6]	Completion queue

continued on next page

continued from previous page		
Type	Name	Description

We define $last(cq)$ and $free(cq)$ as in the case of the **FBPU**.

4.4.3 DU

Inner State

The **DU** has as inner state the number of free rename registers. The following type is defined:

Name	Component	Description
RRUsage	int:3 GPRs	Number of free GPR rename registers
	int:3 FPRs	Number of free FPR rename registers
	int:1 CTRs	Number of free CTR rename registers
	int:1 CRs	Number of free CR rename registers
	int:1 LRs	Number of free LR rename registers

Here int:6 is the type of numbers n with $0 \leq n \leq 6$, and likewise for int:1.

The inner state is an *RRUsage* object telling the number of free rename registers of the various kinds:

Type	Name	Description
RRUsage	rrfree	Number of free rename registers

We define addition and subtraction operations on *RRUsage* in the canonical manner, together with comparison.

For an instruction i we write $rruse(i)$ for the static rename register requirements of that instruction; we write $unit(i)$ for the set of all functional units this instruction may be dispatched to.

State Evolution

The following microsteps are performed on each cycle update by the **DU**.

Step 1: This step handles retirement of instructions

```
If **retired**($rr$) signal received then
    $rrfree \leftarrow rrfree + rr[0] + rr[1]$
Fi
```

Step 2: This step handles a flush after misprediction

```
If **flush** signal received then
  $\forall 0 \leq j \leq last(cq)$ :
    If $cq[j].spec$ then
      $rrfree \leftarrow rrfree + rruse(cq[j].addr)$
    Fi
Fi
```

Step 3: This step handles stalls

```
If **stall** signal received then
  Goto Step 5
Fi
```

Step 4: This step handles dispatch

```
If **IQ**$(a, p)$ signal not received then
  Goto Step 5
Fi
```

Step 4a: Dispatch first instruction

```
$n \leftarrow 0$
If $a[0] \neq \mathsf{none}$ and $free(cq) \neq 0$ and $rruse(a[0]) \leq rrfree$ then
  **Can dispatch first instruction**
  If $a[0]$ is a branch then
    $u \leftarrow \mathsf{branch}$
  Else
    Find first $u$ in $[\mathsf{IU2}, \mathsf{IU1}, \mathsf{LSU}, \mathsf{SRU}, \mathsf{FPU}]$ which satisfies:
      **busy-**$u$ signal not received and $u \in unit(a[0])$
  Fi
  If $u \neq \mathsf{none}$ then
    **Free unit available**
    $duunit[0] \leftarrow u$
    $i[0] \leftarrow last(cq) + 1$
    $duunit[1] \leftarrow \mathsf{none}$
    $i[1] \leftarrow \mathsf{none}$
    $n \leftarrow n + 1$
    $rrfree \leftarrow rrfree - rruse(a[0])$
  Fi
Fi
```

Step 4b: Dispatch second instruction

```
If $n > 0$ and $a[1] \neq \mathsf{none}$ and $\mathit{free}(cq) > 1$ and
    $\mathit{rruse}(a[1]) \leq \mathit{rrfree}$ and $a[0]$ is not completion or
    refetch serializing Then
  **Maybe dispatch second instruction**
  If $a[1]$ is a branch then
    $u \leftarrow \mathsf{branch}$
  Else
    Find first $u$ in $[\mathsf{IU2}, \mathsf{IU1}, \mathsf{LSU}, \mathsf{SRU}, \mathsf{FPU}] \setminus [r[0].\mathit{unit}]$ which satisfies:
      **busy-**$u$ signal not received and $u \in \mathit{unit}(a[1])$
    **Note:** LSU cannot accept two instructions
              although it has two reservation stations.
  Fi
  If $u \neq \mathsf{none}$ then
    **Free unit available**
    $i[1] \leftarrow \mathit{last}(cq) + 2$
    $\mathit{duunit}[1] \leftarrow u$
    $n \leftarrow n + 1$
    $\mathit{rrfree} \leftarrow \mathit{rrfree} - \mathit{rruse}(a[1])$
  Fi
Fi
```

Step 4c: This step performs the dispatch

```
If $n \neq 0$ then
  Emit **reserve**$(i, p, \mathit{duunit})$ signal
  Add $(a[0], p[0], c_0, \mathsf{true}, \mathit{duunit}[0]), \ldots,$
        $(a[n-1], p[n-1], c_{n-1}, \mathsf{true}, \mathit{duunit}[n-1])$ to CQ
  where $c_i$ is $\mathsf{true}$, if the instruction at address $a[i]$
   is a branch, $\mathsf{false}$ otherwise
  **Note: ready** is initially $\mathsf{true}$, but may be set to $\mathsf{false}$ within **CU**.
  Emit **free**$(n)$ signal
Fi
```

Step 5: The End
Noop

4.4.4 IU1, IU2, SRU

These three functional units work in the same way and have the same inner state. Therefore, in this section we write U to denote one of **IU1**, **IU2** or **SRU**.

Inner State

Unit U has the following inner state:

Type	Name	Description
CQIndex	res	Instruction in reservation station
CQIndex	work	Instruction executing in unit
int	cycles	Cycles remaining for executing instruction

We will write $cycles(i)$ for an instruction i to denote the number of cycles, this instruction takes to completely execute.

State Evolution

The following micro steps are executed on each cycle update for U:

Step 1: This step handles a new instruction

```
If **reserve**$(i,p,u)$ signal received then
    If $i[0] \neq$ none and $u[0] = U$ then
        **Dispatch to this unit**
        $res \leftarrow i[0]$
    Else If $i[1] \neq$ none and $u[1] = U$ then
        $res \leftarrow i[1]$
    Fi
Fi
```

Step 2: This step handles a flush signal

```
If **flush** signal received then
    If $res \neq$ none and $cq[res].spec$ then
        **Flushed from CQ, flush it from $U$**
        $res \leftarrow$ none
    Fi
    If $work \neq$ none and $cq[work].spec$ then
        $work \leftarrow$ none
        $cycles \leftarrow 0$
    Fi
Fi
```

Step 3: This step advances reservation to work

```
If $res \neq$ none and $cq[res].ready =$ true and
   $work =$ none then
    $work \leftarrow res$
    $res \leftarrow$ none
    $cycles \leftarrow cycles(cq[work].addr)$
Fi
```

Step 4: This step does some work

```
If $work \neq$ none then
    $cycles \leftarrow cycles - 1$
    If $cycles = 0$ then
        **Done**
        $work \leftarrow$ none
        Emit **done**-$U$ signal
    Fi
Fi
```

Step 5: This step handles the busy condition

```
If $res \neq$ none then
    Emit **busy**-$U$ signal
Fi
```

4.4.5 LSU

Inner State

The following type is defined:

Name	Component	Description
LState	Idle |	Unit is idle
	Await |	Unit waits for datum
	Ignore |	Unit ignores next datum
	Hold	Unit is stopped

The **LSU** has the following inner state:

Type	Name	Description
CQIndex	res[2]	Two reservation stages
CQIndex	ea	Instruction in EA stage
CQIndex	access	Instruction in access stage
Addr	store[3]	The store buffer (store[0] is the next to execute)
CQIndex	sIns[3]	Instruction corresponding to store buffer entry
int	sPending	Number of store and sIns entries
continued on next page		

continued from previous page		
Type	Name	Description
LState	state	State of the access machinery
int	numAcc	Number of accesses instruction will perform

State Evolution

LSU performs the following micro steps in each cycle update:

Step 1: This step handles newly dispatched instructions

```
If $\textbf{reserve}(i,p,u)$ signal received and $\{u[0],u[1]\} \cap \{\textbf{LSU}\} \neq \emptyset$ then
  Dispatch for us
  If $u[0] = \textbf{LSU}$ then
    $j \leftarrow i[0]$
  Else
    $j \leftarrow i[1]$
  Fi
  If $res[0] = \mathsf{none}$ then
    $res[0] \leftarrow j$
  Else
    $res[1] \leftarrow j$
  Fi
Fi
```

Step 2: This step handles a flush signal

```
If flush signal received then
  If $access \neq \mathsf{none}$ and $cq[access].spec$ then
    If $state =$Await then
      $state \leftarrow$ Ignore
    Else
      $state \leftarrow$Idle
    Fi
    $access \leftarrow \mathsf{none}$
  Fi
  If $ea \neq \mathsf{none}$ and $cq[ea].spec$ then
    $ea \leftarrow \mathsf{none}$
  Fi
  If $res[1] \neq \mathsf{none}$ and $cq[res[1]].spec$ then
    $res[1] \leftarrow \mathsf{none}$
  Fi
  If $res[0] \neq \mathsf{none}$ and $cq[res[0]].spec$ then
```

```
    $res[0] \leftarrow$ none
  Fi
  $\forall 3 \geq j \geq 0$ :
    If $sIns[j] \neq$ none and $cq[sIns[j]].spec$ then
      $sIns[j] \leftarrow$ none
      $store[j] \leftarrow$ none
    Fi
Fi
```

Step 3: This step handles the resolve signal

```
If **resolve** signal received then
  If $state =$ Hold then
    $state \leftarrow$ Idle
  Fi
Fi
```

Step 4: This step handles pipeline advance from reservation

```
If $ea =$ none and $res[0] \neq$ none and $cq[res[0]].ready =$ true then
  $ea \leftarrow res[0]$
  $res[0] \leftarrow res[1]$
  $res[1] \leftarrow$ none
Fi
```

Step 5: This step handles delayed stores

```
If **finished** signal received then
  $sPending \leftarrow sPending + 1$
Fi
If $state =$ Idle and **working** signal not received and
   $sPending > 0$ then
  Emit **store**$((a,l))$ signal, where $a = store[0]$
  $sPending \leftarrow sPending - 1$
  $store[0] \leftarrow store[1]$
  $store[1] \leftarrow store[2]$
  $store[2] \leftarrow$ none
  $sIns[0] \leftarrow sIns[1]$
  $sIns[1] \leftarrow sIns[2]$
  $sIns[2] \leftarrow$ none
  Goto Step 7
Fi
```

Step 6: This step processes accesses

```
If $access$ = none then
    Goto Step 7
Fi
```

Step 6a: This step handles a read

```
If $access$ is not a read access then
    Goto Step 6c
Fi
```

Step 6b:

```
If $state$ = Idle and **working** signal not received then
    Let $a$ be the address of the access
    If $a$ is guarded and $cq[access].spec$ then
        **Speculative access to guarded space**
        $state \leftarrow$ Hold
    Else
        $state \leftarrow$ Await
        Emit **read**($a$) signal
    Fi
Else If $state$ = Await and **data** signal received then
    $numAcc \leftarrow numAcc - 1$
    If $numAcc = 0$ then
        $access \leftarrow$ none
        Emit **done-LSU** signal
    Fi
Else If $state$ = Ignore and **data** signal received then
    $state \leftarrow$ Idle
Fi
Goto Step 7
```

Step 6c: This step handles a write access

```
If $access$ is a single store then
    If $sPending < 3$ then
        $store[sPending] \leftarrow addr(access)$
        $sIns[sPending] \leftarrow access$
        $sPending \leftarrow sPending + 1$
        Emit **done-LSU** signal
        $access \leftarrow$ none
    Fi
Else
    **Multiple word/string store**
    If **working** signal not received then
        Emit **store**$(a)$ signal, where $a = addr(access) + c$
            where $c$ models the progress through the word/string
        $numAcc \leftarrow numAcc - 1$
        If $numAcc = 0$ then
            $access \leftarrow$ none
            Emit **done-LSU** signal
        Fi
    Fi
Fi
```

Step 7: This step handles pipeline advance to access

```
If $access =$ none and $ea \neq$ none then
    $access \leftarrow ea$
    $ea \leftarrow$ none
    If $access$ is a multiple load/store then
        $numAcc \leftarrow$'# of accesses'
    Else
        $numAcc \leftarrow 1$
    Fi
Fi
```

Step 8: This step handles the busy signal

```
If $res[1] \neq$ none then
    Emit **busy-LSU** signal
Fi
```

4.4.6 FPU

Inner State

The **FPU** contains the following inner state:

Type	Name	Description
CQIndex	res	Reservation station
CQIndex	work[3]	Stages
int	cycles[3]	Cycles remaining in stages
Bool	block	We are executing a blocking instruction

State Evolution

The **FPU** performs the following micro steps on each cycle update:

Step 1: This step handles new instructions

```
If **reserve**($i$,$p$,$u$) signal received and $\{u[0], u[1]\} \cap \{\textbf{FPU}\} \neq \emptyset$ then
  If $u[0] = \textbf{FPU}$ then
    $res \leftarrow i[0]$
  Else
    $res \leftarrow i[1]$
  Fi
Fi
```

Step 2: This step handles the flush signal

```
If **flush** signal received then
  If $res \neq \textsf{none}$ and $cq[res].spec$ then
    $res \leftarrow \textsf{none}$
  Fi
  $\forall 0 \leq j \leq 2$ :
    If $work[j] \neq \textsf{none}$ and $cq[work[j]].spec$ then
      $work[j] \leftarrow \textsf{none}$
      $cycles[j] \leftarrow 0$
      If $j = 0$ then
        $block \leftarrow \textsf{false}$
      Fi
    Fi
Fi
```

Step 3: This step advances the reservation station

```
If $res \neq \textsf{none}$ and $cq[res].ready = \textsf{true}$ and
    $work[0] = \textsf{none}$ then
  $work[0] \leftarrow res$
  $res \leftarrow \textsf{none}$
  If $isblocking(cq[work[0]].addr)$ then
    **A blocking instruction**
    $block \leftarrow \textsf{true}$
  Fi
  $cycles[0] \leftarrow cycles(cq[work[0]].addr)$
Fi
```

Step 4: This step does some work

```
$\forall 0 \leq j \leq 2:$
    If $work[j] \neq \mathsf{none}$ then
        $cycles[j] \leftarrow \max(0, cycles[j] - 1)$
    Fi
```

Step 5: This step advances the pipeline

```
If $work[2] \neq \mathsf{none}$ and $cycles[2] = 0$ then
    Emit **done-FPU** signal
    $work[2] \leftarrow \mathsf{none}$
Fi
If $work[1] \neq \mathsf{none}$ and $cycles[1] = 0$ and $work[2] = \mathsf{none}$ then
    $work[2] \leftarrow work[1]$
    $work[1] \leftarrow \mathsf{none}$
    $cycles[2] \leftarrow fpucycles(3, cq[work[2]].addr)$
Fi
If $work[0] \neq \mathsf{none}$ and $cycles[0] = 0$ and $work[1] = \mathsf{none}$ and
  $block = \mathsf{false}$ then
    $work[1] \leftarrow work[0]$
    $work[0] \leftarrow \mathsf{none}$
    $cycles[1] \leftarrow fpucycles(2, cq[work[1]].addr)$
Fi
```

Step 6: This step handles finish of blocking instruction

```
If $block$ and $cycles[0] = 0$ and $work[1] = work[2] = \mathsf{none}$ then
    $block \leftarrow \mathsf{false}$
    $work[0] \leftarrow \mathsf{none}$
    Emit **done-FPU** signal
Fi
```

Step 7: This step handles the busy signal

```
If $res \neq \mathsf{none}$ then
    Emit **busy-FPU** signal
Fi
```

4.4.7 CU

Inner State

The **CU** only uses the **CQ** as state.

State Evolution

We write $gprups(i)$ to denote the number of GPR updates of instruction i; likewise with $fprups(i)$ and FPR updates.

These micro-steps are performed on each cycle update by **CU**:

Step 1: This step handles finished instructions

```
If **done-$U$** signal received then
  Let $j = \min_{\{0,\ldots,last(cq)\}} : cq[j].complete = \mathsf{false}$ and $cq[j].unit = U$
    $cq[j].complete \leftarrow \mathsf{true}$
Fi
```

Step 2: This step handles a flush signal

```
If **flush** signal received then
  $\forall 0 \leq j \leq last(cq)$ :
    If $cq[j].spec = \mathsf{true}$ then
      $cq[j] \leftarrow (\mathsf{none}, \mathsf{true}, \mathsf{false}, \mathsf{false}, \mathsf{none})$
      **Note:** The *spec* field remains true because it is still accessed.
    Fi
Fi
```

Step 3: This step handles a resolve signal

```
If **resolve** signal received then
  $\forall 0 \leq j \leq last(cq)$ :
    $cq[j].spec \leftarrow \mathsf{false}$
Fi
```

Step 4: This step handles serialization

```
$\forall 0 \leq j \leq last(cq)$ :
  $cq[j].ready \leftarrow \mathsf{true}$
  If $cq[j]$ is execution serialization and $j \neq 0$ then
    **Cannot execute if there is a predecessor in CQ**
    $cq[j].ready \leftarrow \mathsf{false}$
  Fi
  If $cq[j]$ is completion serialization then
    **Cannot dispatch another instruction**
    Emit **stall** signal
  Fi
```

Step 5: This step handles stalls due to operands

```
$\forall 0 \le j \le last(cq)$ :
  If $cq[j].ready = \textsf{true}$ then
    $\forall o \in operands(cq[j])$ :
      $k \leftarrow \max_{\{0,\ldots,j-1\}} : o \in write(k)$
      If $k \ne \textsf{none}$ and $cq[k].complete = \textsf{false}$ then
        $cq[j].ready \leftarrow \textsf{false}$
      Fi
  Fi
```

Step 6: This step handles retirement

```
$n \leftarrow 0$
If $cq[0].addr \ne \textsf{none}$ and $cq[0].complete$ then
  **Note:** $cq[0]$ cannot be speculative; it cannot depend on previous instructions.
  $n \leftarrow 1$
  $rr \leftarrow rruse(cq[0])$
  If $cq[1].addr \ne \textsf{none}$ and $cq[1].complete$ then
    **Note:** Here $cq[1]$ cannot be speculative either.
    If $cq[1]$ is an integer or load instruction and
       $gprups(cq[0]) + gprups(cq[1]) \le 2$ and
       $fprups(cq[0]) + fprups(cq[1]) \le 2$ then
      **We can retire two instructions**
      $n \leftarrow 2$
      $rr \leftarrow rr + rruse(cq[1])$
    Fi
  Fi
Fi
If $n > 0$ then
  Emit **retired**$(rr)$ signal
  If $cq[0], \ldots, cq[n-1]$ contain a single store instruction then
    Emit **finished** signal
  Fi
  Remove $cq[0], \ldots, cq[n-1]$ from CQ
Fi
```

4.4.8 BU

Inner State

The following types are defined:

Name	Component	Description
IBCState	HH |	Instruction fetch is a hit-hit
	HM |	Fetch is a hit-miss
	MH |	Fetch is a miss-hit
	MM	Fetch is a miss-miss
ACSource	D |	Access is a data read
	S |	Access is a store or cache line flush
	I |	Access is an instruction fetch
	N	No access
Schedule	ts(Addr a, Bool isburst) |	Start access by emitting a **TS** signal
	aack |	Wait for **AACK** signal
	emit(D) |	Emit a **data** signal
	emit(F) |	Emit a **fetched** signal
	ta	Wait for a **TA** signal
Access	ACSource src	The source of the access
	Addr CL	The cacheline affected (if any)
	Schedule schedule[]	The schedule for the access

The bus unit contains the following inner state:

Type	Name	Description
Addr	IBAddr	Instruction address in instruction buffer or none
int:3	IBLen	# of instructions to fetch
IBCState	IBC	Cache behavior of instruction fetch
Bool	IBSched	Has fetch been put into acc?
Addr	DBAddr	Data read address in data buffer or none
int	DBLen	Length of data access in data buffer
Bool	DBSched	Has read been put into acc?
Addr	SBAddr	Data store address in store buffer or none
int	SBLen	Length of store in store buffer
Bool	SBSched	Has write been put into acc?
Access	acc[4]	Two pipelineable accesses and two split accesses

The state of the bus unit consists of four groups:

- The *Instruction Buffer* (IB) holds a request for an instruction fetch from the FBPU. The request remains in the IB until the instruction fetch has completed and the instructions have been returned to the FBPU. The IB is filled by a **fetch(AddrLen)** signal. It has four components: the address to fetch from, the number of instructions to fetch, the cache behavior and if it has been put into `acc` already. The cache behavior determines, if the instruction fetch is a miss or hit for the two lines that may be involved. If the instruction fetch only spans one cache line (or none, if instructions are not fetched from cacheable memory), the other components are assumed to be 'hits'. I.e. an access covering two lines, where the first line is a cache miss and the second a cache hit would have a cache behavior classification of 'MH'. An access that covers one line and misses in that line would also be classified as 'MH'. The case 'HH' means that all line accesses hit in the cache, but that one of the two possible lines momentarily completes loading over the bus (from a previous access). Such an access has to wait until the current bus access has completed.

- The *Data Buffer* (DB) holds a read request from the LSU. It is filled upon receipt of a **read(AddrLen)** signal from the LSU. It contains three components: the address and length of the data to be read and if the read has already been put into `acc`.
- The *Store Buffer* (SB) holds a data write request from the LSU. It is filled upon receipt of a **store(AddrLen)** signal from the LSU and has the same components as the DB.
- Finally the *Access Slots* (AC) give the accesses that are scheduled to be performed over the bus. The PPC755 can pipeline two accesses, i.e. two accesses may be active at the same time, overlaping address and data phases. Every access (Access) contains its source (DB, SB, IB or empty, if the slot is not used). In addition, the cacheline affected by the access, if it is a cache line fill is recorded. Finally, the schedule for the access is recorded. A schedule contains a sequence of items:
  - **ts(a, b)** indicates that a **TS** signal is to be emitted to the **CSU**. The parameters for the signal are taken form the **src** attribute of the access and the **a** and **b** parameters of the item.
  - **aack** the access has to wait for the **AACK** signal.
  - **emit(D)** a **data** signal is to be emitted.
  - **emit(F)** a **fetched** signal is to be emitted. The parameter to the signal depends on the alignment of the access in the **IB**.
  - **ta** wait for an **TA** signal.

Since an instruction fetch in **IB** may have to be split in up to three non interruptible accesses (in the case that a fetch of four instructions starts at an address not dividable by 8), the two "additional" accesses have to be kept in **acc[2,3]**. This is just to ensure that the accesses are not interrupted by other accesses.

State Evolution

Some assumptions have been made in the model of the bus unit:

- The data cache is only used in write-through mode. I.e. stores cannot affect the cache contents.
- Data accesses do not cross cache line boundaries. This would violate the alignment restrictions of the PowerPC anyhow.
- When multiple requests are queued up at the bus unit, they are processed in the order: stores, data reads, instruction fetches.

We write $cclass(a)$ to denote the access classification of an address a (HH, HM, MH, or MM).

We write $insinline(a,l)$ for the number of instructions in the cache line which contains the address a, where l is the total number of instructions.

The function $getsched(a,l,c)$ returns a vector of `Access` for the instruction fetch from address a. The parameter l gives the number of instructions to fetch, the boolean parameter c is true if the access is cacheable. The function is defined by:

$$getsched(a,l,\mathsf{false}) = \begin{cases} [(I,\mathsf{none},s(a,1))] & \text{, if } a,l \text{ span 1 double word} \\ [(I,\mathsf{none},s(a,2))] & \text{, if } a,l \text{ span 2 double words} \\ [(I,\mathsf{none},s(a,3))] & \text{, if } a,l \text{ span 3 double words} \end{cases}$$

Where $s(a,n)$ is defined by

$$s(a,n) = \begin{cases} [\mathbf{ts}(a,\mathsf{false}),\mathbf{aack},\mathbf{ta},\mathbf{emit}(F)] & \text{, if } s=1 \\ [\mathbf{ts}(a,\mathsf{false}),\mathbf{aack},\mathbf{ta},\mathbf{emit}(F)].s(al(a),n-1) & \text{, if } s>1 \end{cases}$$

Where $al(a)$ is the address a aligned to the beginning of the next double word.

$$getsched(a,l,\mathsf{true}) = \begin{cases} [(I,a,t(a,l))] & \text{, if 1 cache line access} \\ [(I,a,t(a,insinline(a))), & \\ \;(I,a',t(a',insinline(a')))] & \text{, else; with} \\ & a' = a + 4insinline(a) \end{cases}$$

Where $t(a,l)$ is defined by

$$t(a,l) = \begin{cases} [\mathbf{ts}(a,t),\mathbf{aack},t',\mathbf{ta},\mathbf{ta},\mathbf{ta}] & \text{, if } l < ins(a) \\ [\mathbf{ts}(a,t),\mathbf{aack},t',t',\mathbf{ta},\mathbf{ta}] & \text{, if } l < ins(a)+2 \\ [\mathbf{ts}(a,t),\mathbf{aack},t',t',t',\mathbf{ta}] & \text{, else} \end{cases}$$

Where $t' = \mathbf{ta},\mathbf{emit}(F)$ and $ins(a)$ is the number of instructions in the double word containing a.

The BU evolves in 8 steps:

Step 1: Advance signaled fetch into IB

```
If **fetch**$(a,l)$ signal received then
    $(IBAddr, IBLen, IBSched) \leftarrow (a, l, \mathsf{false})$
    $IBC \leftarrow \mathsf{HH}$
    **Determine cache behavior**
    If memory area at $IBAddr$ is cacheable then
        $IBC \leftarrow cclass(IBAddr)$
        Update the instruction cache
        If access is in one cache line only then
            If $IBC = HH$ then
                If $IBAddr$ does not clash with access in $acc[0..3]$ then
                    Emit **fetched**$(IBLen)$ signal, clear IB
                Fi
            Fi
        Else **Two line access**
            $n \leftarrow 0$
            If $IBC \in \{\mathsf{HH}, \mathsf{HM}\}$ then
                If first line does not clash with $acc[0..3]$ then
                    $n \leftarrow insinline(IBAddr, IBLen)$
                    If $IBC = \mathsf{HH}$ and
                            second line does not clash with $acc[0..3]$ then
                        $n \leftarrow n + insinline(IBAddr + 4 * n, IBLen - n)$
                    Fi
                Fi
            Fi
            $IBAddr \leftarrow IBAddr + 4 * n$
            $IBLen \leftarrow IBLen - n$
            If $0 = IBLen$ then
                Clear IB
            Fi
            If $0 \neq n$ then
                Emit **fetched**$(n)$ signal
                $IBC \leftarrow tail(IBC)$
            Fi
        Fi
    Fi
Fi
```

Step 2: Advance signaled store to SB

```
If **store**$(a,l)$ signal received then
```

```
    $(SBAddr, SBLen, SBSched) \leftarrow (a, l, \mathsf{false})$
    If the access is to write-back area then **write-back, no replacement**
        Clear SB **Returns immediately**
    Fi
Fi
```

Step 3: Advance signaled read to DB

```
If **read**$(a, l)$ signal received then
    $(DBAddr, DBLen, DBSched) \leftarrow (a, l, \mathsf{false})$
    If access is cacheable then
        If $l = 0$ then **Cache line invalidation by `dcbi`**
            Remove cache line for $DBAddr$ from cache
            If $DBAddr$ does clash with $acc[0..3]$ then
                $DBLen \leftarrow 0$
            Else
                Emit **data** signal
                Clear DB
            Fi
            Goto step 4
        Else if $l = 32$ then **cache line zero by `dcbz`**
            If $DBAddr$ does clash with $acc[0..3]$ then
                $DBLen \leftarrow 0$
            Else
                Emit **data** signal
                Clear DB
            Fi
            Goto step 4
        Fi
        $cata \leftarrow cclass(DBAddr)$
        Update the data cache
        If access clashes with data cache line fill in $acc[0..3]$ then
            $DBLen \leftarrow 0$ **Can return as soon as fill completes**
        Else If $cata = \mathsf{H}$ then
            Emit **data** signal, Clear DB
        Fi
    Fi
Fi
```

Step 4: Advance store to access

```
If $SBAddr \neq \mathsf{none}$ and $free(acc)$ and $!SBSched$ then
    If $32 = SBLen$ then **dcbf instruction cache line flush**
    insert $(S, \text{none}, [\mathbf{ts}(SBAddr, \text{true}), \mathbf{aack}, \mathbf{ta}, \mathbf{ta}, \mathbf{ta}, \mathbf{ta}])$ into $acc$
  Else **single store**
    Insert $(S, \text{none}, [\mathbf{ts}(SBAddr, \text{false}), \mathbf{aack}, \mathbf{ta}])$ into $acc$
  Fi
  $SBSched \leftarrow \text{true}$
Fi
```

Step 5: Advance read to access

```
If $DBAddr \neq \text{none}$ and $free(acc)$ and $!DBSched$ then
  If $DBAddr$ is cacheable then
    If $DBAddr$ does not clash with $acc$ then
      Insert $(D, DBAddr,$
        $[\mathbf{ts}(DBAddr, \text{true}), \mathbf{aack}, \mathbf{ta}, \mathbf{emit}(D), \mathbf{ta}, \mathbf{ta}, \mathbf{ta}])$ into $acc$
    Fi
  Else **Single access**
    insert $(D, \text{none}, [\mathbf{ts}(DBAddr, \text{false}), \mathbf{aack}, \mathbf{ta}, \mathbf{emit}(D)])$ into $acc$
  Fi
  $DBSched \leftarrow \text{true}$
Fi
```

Step 6: Advance fetch to access

```
If $IBAddr \neq \text{none}$ and $free(acc)$ and $!IBSched$ then
  If $IBAddr$ is cacheable then
    If $(IBAddr, IBLen)$ is in one cache line then
      If $IBC = \text{MH}$ then **HH is a noop, HM, MM impossible**
        $scheds[] \leftarrow getsched(IBAddr, IBLen, \text{true})$
        insert $scheds$ into $acc$
        $IBSched \leftarrow \text{true}$
      Fi
    Else **Two cachelines**
      $scheds \leftarrow \text{empty}$
      case $IBC$ of
        HH: **line clash, noop**
        HM: **first line clash**
          $n \leftarrow insinline(IBAddr, IBLen)$
          $scheds \leftarrow getsched(IBAddr + 4 * n, IBLen - n, \text{true})$
        MH: **first miss, second will be hit**
          $scheds \leftarrow getsched(IBAddr, insinline(IBAddr, IBLen), \text{true})$
        MM: **two misses**
          $scheds \leftarrow getsched(IBAddr, IBLen, \text{true})$
      esac
      If $scheds \neq$ empty then
        Insert $scheds$ into $acc$
        $IBSched \leftarrow$ true
      Fi
    Fi
  Else **Uncacheable access**
    $scheds \leftarrow getsched(IBAddr, IBLen,$ false$)$
    Insert $scheds$ into $acc$
    $IBSched \leftarrow$ true
  Fi
Fi
```

Step 7: Process accesses

```
If $acc[1].src \neq N$ then **A 2nd access there**
  If $hd(acc[1]) = \mathbf{ts}(a,b)$ and $hd(acc[0]) \notin \{\mathbf{aack}, \mathbf{ts}(a',b')\}$ and
      the bus/core clock are aligned then
    emit $\mathbf{TS}(a, acc[1].src, l)$ signal, where
    $l = \begin{cases} 4 & \text{, if } b = \text{true} \\ 0 & \text{, if } acc[1].src = S \text{ and } SBLen < 8 \\ 1 & \text{, else} \end{cases}$
    $acc[1].schedule \leftarrow tail(acc[1].schedule)$
  Fi
Fi
If $acc[0].src \neq N$ then **1st access not empty**
  If $hd(acc[0].schedule) = \mathbf{ts}(a,b)$ and core/bus clock are aligned then
    **Start transfer**
    emit $\mathbf{TS}(a, acc[0].src, l)$ signal, where $l$ is defined as above.
    $acc[0].schedule \leftarrow tail(acc[0].schedule)$
  Fi
  If **AACK** signal received then
    If $hd(acc[0].schedule) = \mathbf{aack}$ then
      $acc[0].schedule \leftarrow tail(acc[0].schedule)$
    Else
      $acc[1].schedule \leftarrow tail(acc[1].schedule)$
    Fi
  Fi
  If **TA** signal received then **1st access must wait for it**
    $acc[0].schedule \leftarrow tail(acc[0].schedule)$
  Fi
  If $hd(acc[0].schedule) = \mathbf{emit}(D)$ then
    Emit **data** signal
    Clear DB
    $acc[0].schedule \leftarrow tail(acc[0].schedule)$
  Fi
  If $hd(acc[0].schedule) = \mathbf{emit}(F)$ then
    $n \leftarrow ins(IBAddr)$
    Emit **fetched**$(n)$ signal
    $IBLen \leftarrow IBLen - n$
    $IBAddr \leftarrow IBAddr + 4n$
    If $IBLen = 0$ then
      Clear IB
    Fi
    $acc[0].schedule \leftarrow tail(acc[0].schedule)$
  Fi
  If $acc[0].schedule = []$ then **access finished**
    case $acc[0].src$ of
      S:
      Clear SB
      D:
      If $acc[0].CL \neq$ none and $DBAddr$ clashes with $acc[0].CL$ and
          $!DBSched$ then
        **DB waits for this access**
        Clear DB
        Emit **data** signal
      Fi
      I:
      If $acc[0].CL \neq$ none and $IBAddr$ clashes with $acc[0].CL$ and
          $!IBSched$ then
        $n \leftarrow 0$
        If $IBC = HH$ then
          $n \leftarrow IBLen$
        Else If $IBC = HM$ then
          $n \leftarrow insinline(IBAddr, IBLen)$
        Fi
        If $n \neq 0$ then
          Emit **fetched**$(n)$ signal
          $IBLen \leftarrow IBLen - n$
          $IBAddr \leftarrow IBAddr + 4n$
          $IBC \leftarrow tail(IBC)$
          If $IBLen = 0$ then
            Clear IB
          Fi
```

```
|   |   |   |   Fi
|   |   |   Fi
|   |   esac
|   |   Move $acc[1..3]$ up one slot
|   |   Clear $acc[3]$
|   Fi
Fi
```

Step 8: Emit Working signal

```
If $SBAddr \neq \mathsf{none}$ then
|   Emit **working** signal
Fi
```

4.4.9 CSU

This unit has been left out for space considerations.

4.4.10 State Predicates

Finally, we have to define the state predicates from Definition 4.1.3. We thus introduce new component variables, *ppcRetired*, *ppcHalted*, *ppcNext*, *ppcBranches* and *ppcNops*:

- *ppcRetired* is a sequence of addresses which denote the instructions already retired. This accounts for multiple retirement. We update this sequence whenever an instruction retires from the CU.
- *ppcHalted* is a Boolean that records the fact that the last instruction of the program has been executed
- *ppcNext* records the address of the next instruction to be executed
- *ppcBranches* is an integer that counts the number of branches dropped or folded without an entry in the CQ
- *ppcNops* is an integer that counts the number of noop instructions dropped by the DU

The state predicates are more complicated to define for the PPC 755 because instructions may be dropped without passing through the CQ or CU: branches and noops. Therefore, we record the number of these instructions in the two variables *ppcNops* and *ppcBranches*. They are increased in the FBPU and DU if such an instruction is dropped or folded. In the CU, it is checked if the actual instruction is such a noop or branch and retirement is adjusted accordingly and the variables are decremented. Then we can define

$$\mathcal{F}(n,s) := hd(s(ppcRetired)) = n$$

and

$$\mathcal{R}(n,s) := s[ppcRetired \leftarrow tl(s(ppcRetired))]$$

ppcNext is set to the address of the next instruction as soon as that instruction is clear, i. e. after speculative branches have been resolved. As for the ColdFire version of this predicate, the address is computed by a function $target(acti)$ for the actual instruction. The update of this variable is at the begining for non branch instructions and in the FBPU after the branch resolves. Then we can define

$$\mathcal{N}(n,s) := n = s(ppcNext)$$

Finally, *ppcHalted* is set as soon as the last instruction retires in the CU and we have

$$\mathcal{H}(s) := s(ppcHalted)$$

Chapter 5

Pipeline Analysis

If you try to know everything, you will learn nothing.

Demokrit

The goal of our pipeline analysis is to find a correct approximation to the WCET of a program defined by Equation (4.1.6). We do this by finding an abstraction of the collecting semantics defined in Equation (4.1.5). Since the implementation of our analysis is a data-flow analysis, finding an abstraction for the collecting semantics boils down to finding abstract versions $\hat{\mathcal{T}_e}$ of the transfer functions $\mathcal{T}_e'$ associated with the edges in the CFG that satisfy Equation (3.3.4) or (3.3.6).

First, the abstract domain of the analysis has to be determined and the relation of abstract values of this domain to sets of concrete states has to be defined. An abstract value $\hat{\sigma} \in \hat{\Sigma}$ is a pair $(\check{s}, \hat{m})$, where $\check{s}$ is a set of abstract states $\hat{s} \in \hat{\mathcal{S}}$, each of which represents a set of concrete states s. $\hat{m}$ is an upper bound for the times of all executions ending in one of the concrete states s. Thus, the abstract domain is $\hat{\Sigma} = \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z}$.

Additionaly, we have a concretization function $\Gamma : \hat{\mathcal{S}} \to \mathcal{P}(\mathcal{S})$ which gives the set of concrete states represented by an abstract state $\hat{s}$. We will later see, how this Γ is constructed from the definitions of the state components and their respective abstraction. Unfortunately, it has shown to be difficult to define a *Galois Connection* between the domains $\mathcal{P}(\Sigma)$ and $\hat{\Sigma}$ simply because Γ does not allow to define a corresponding abstraction function $\alpha' : \mathcal{P}(\mathcal{S}) \to \hat{\mathcal{S}}$. This is because, e. g. for the PowerPC cache, there does not exist a *best* abstraction for a given set of concrete components, cf. Example 3.2.27 in Section 3.2.1. Thus, we cannot use condition (3.3.4) to prove the soundness of an abstraction. Since the presence of a concretization function with certain properties is sufficient to show correctness by using (3.3.6), we will present the proof in this way.

In the following we will first determine what abstract entities have to be defined and how they must be related to their concrete counterparts. Using this we give a *global correctness* proof that only relies on a property of the state transfer function $\mathbb{T}$ and its abstract counterpart $\hat{\mathbb{T}}$.

Then we will give an abstraction for our abstract version $\hat{\mathbb{T}}$ of $\mathbb{T}$ by providing abstract unit updates and component domains. By proving that our definition of $\hat{\mathbb{T}}$ has the desired properties required by the global correctness proof, the correctness of our abstraction is shown.

This approach allows to prove other implementations of the pipeline analysis correct, e. g. the cycle update can be formulated in the synchronous language Esterel ([Ber]) by simply showing that the correctness conditions between $\mathbb{T}$ and the Esterel cycle update $\hat{\mathbb{T}}_E$ hold.

In the remainder of this chapter, we will first give some notational definitions that will be used in the sequel. Then we will give the general correspondence between the concrete domain and the abstract one, $\hat{\Sigma}$, based on a concretization function $\Gamma : \hat{\mathcal{S}} \to \mathcal{P}(\mathcal{S})$. With this correspondence we give minimal relations between concrete and abstract predicates and functions, including $\mathbb{T}$. In Section 5.2 we will show that any analysis satisfying these relations is correct w.r.t. the coarse semantics. After this, we will give in Section 5.3 abstractions for our definition of $\mathbb{T}$ in terms of unit updates. The proof that these abstractions satisfy the general abstraction requirements establishes the correctness of our analysis using unit updates.

5.1 Notation

Given a pair (a,b) of values, we define the functions

$$\begin{array}{rcl} \mathrm{fst}(a,b) & = & a \\ \mathrm{snd}(a,b) & = & b \end{array}$$

We extend these functions to set of pairs, e. g. $\mathrm{fst}(A) = \{\mathrm{fst}(a,b) \mid (a,b) \in A\}$. We abbreviate a tupel $(a_1,\dots,a_n)$ by writing $\vec{a}$. We write $(\vec{a},\vec{b})$ for the tupel $(a_1,\dots,a_n,b_1,\dots,b_m)$.

Let $\mathcal{Z}$ be a finite interval of $\mathbb{N}$ containing all possible execution times: $\mathcal{Z} = \{0,1,\dots,T_{\max}\}$. $\mathcal{Z}$ together with the usual ordering $\leq$ on natural numbers forms a complete lattice with least element 0, greatest element $T_{\max}$ and $\bigsqcup_{\mathcal{Z}} N = \max N$, $\bigsqcap_{\mathcal{Z}} N = \min N$. We define an addition $+$ on $\mathcal{Z}$ as the usual addition on natural numbers, saturated at the value $T_{\max}$, i. e. the sum in $\mathcal{Z}$ of two numbers will be $T_{\max}$ if the sum of the corresponding natural numbers is larger.

Let the set $\mathcal{P}(\mathcal{S} \times \mathcal{Z})$ be ordered by $\subseteq$. Then $\mathcal{P}(\mathcal{S} \times \mathcal{Z})$ is a complete lattice with least element $\emptyset$, greatest element $\mathcal{S} \times \mathcal{Z}$ and $\bigsqcup = \bigcup$ and $\bigsqcap = \bigcap$.

For a set $\hat{\mathcal{S}}$ let $\mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z}$ be ordered by $\sqsubseteq_\times$, defined by

$$(\hat{S}_1, n_1) \sqsubseteq_\times (\hat{S}_2, n_2) \text{ iff } \hat{S}_1 \subseteq \hat{S}_2 \wedge n_1 \leq n_2$$

Then it is a complete lattice with least element $(\emptyset, 0)$, greatest element $(\hat{\mathcal{S}}, T_{\max})$ and

$$\begin{array}{rcl} \bigsqcup_\times N & = & (\bigcup \mathrm{fst}(N), \max \mathrm{snd}(N)) \\ \bigsqcap_\times N & = & (\bigcap \mathrm{fst}(N), \min \mathrm{snd}(N)) \end{array}$$

Elements $\check{s}, \check{t}$ are always from the set $\mathcal{P}(\hat{\mathcal{S}})$, i. e. sets of abstract pipeline states. $\hat{s}$ and $\hat{t}$ stand for abstract pipeline states from $\hat{\mathcal{S}}$. We write $\hat{m}$, $\hat{n}$ and $\hat{o}$ for elements from $\mathcal{Z}$, if we want to hint that they come from a pair, e. g. $(\check{s}, \hat{m})$, of the abstract domain. We write s, t for concrete states and likewise m, o, p for concrete execution time bounds from $\mathcal{Z}$. Furthermore, we write $\hat{S}$ for a set of pairs from the abstract domain, $\hat{S} \subseteq \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z}$. A variable b is an element from the set Bool; z is from the set Int. Abstract values from a domain are written as $\hat{v}$ or $\hat{v}_i$. Concrete values are written v or v_i.

5.2 Global Correctness

For our pipeline analysis we have to specify abstract counterparts of the concrete domains and semantic functions. We will represent a set of concrete pipeline states from $\mathcal{S}$ by one *abstract pipeline state* $\hat{s}$ from a set $\hat{\mathcal{S}}$. A set of such abstract states, $\check{s} \in \mathcal{P}(\hat{\mathcal{S}})$, will be used as the counterpart for the set $\mathcal{P}(\mathcal{S})$, the domain of the sets of concrete pipeline states.

That is, the concretization function $\Gamma : \hat{\mathcal{S}} \rightarrow \mathcal{P}(\mathcal{S})$ maps one abstract state to the concrete states that are represented by it. This Γ does not need to fulfill further constraints, like monotonicity, etc. In fact, we do not even define an ordering on the set $\hat{\mathcal{S}}$. For different analyses, different version of Γ can be used. In Section 5.3, we will give the Γ that corresponds to the abstractions on unit states defined there.

The domain of the semantics, $\Sigma = \mathcal{S} \times \mathcal{Z}$, consists of pairs of a concrete pipeline state and the execution time of the program when it reaches this state. To obtain an analysis that correctly describes all executions of a program, we have to find an approximation to the collecting semantics, i. e. we have to abstract elements from $\mathcal{P}(\mathcal{S} \times \mathcal{Z})$. We do this by choosing the abstract domain $\hat{\Sigma} = \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z}$, where an element of the abstract domain is a pair $(\check{s}, \hat{m})$ of a set of abstract states and an upper bound to execution times. The meaning of such a pair is the set of concrete pairs, where each concrete state in the pair is described by an abstract state and the execution times of all concrete pairs are at most as large as the upper bound. More formally, we can give a concretization function γ that gives all concrete pairs described by one element from $\hat{\Sigma}$:

$$\begin{aligned} &\gamma : \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z} \to \mathcal{P}(\mathcal{S} \times \mathcal{Z}) \\ &\gamma(\check{s}, \hat{m}) = \bigcup\{(s,m) \mid s \in \Gamma(\hat{s}) \wedge 0 \leq m \leq \hat{m} \wedge \hat{s} \in \check{s}\} \end{aligned} \tag{5.2.1}$$

This γ satisfies the correctness prerequisites of Theorem 3.3.5:

Lemma 5.2.2 (Monotonicity of γ): γ is monotone. **Proof:**
Let $(\check{s}_1, \hat{m}_1) \sqsubseteq_\times (\check{s}_2, \hat{m}_2)$, i. e. we have $\check{s}_1 \subseteq \check{s}_2 \wedge \hat{m}_1 \leq \hat{m}_2$. Then

$$\begin{aligned} & (s,m) \in \gamma(\check{s}_1, \hat{m}_1) \\ \Rightarrow\ & \exists \hat{s} \in \check{s}_1 : s \in \Gamma(\hat{s}) \wedge m \leq \hat{m}_1 \\ \Rightarrow\ & \hat{s} \in \check{s}_2 \wedge m \leq \hat{m}_1 \leq \hat{m}_2 \\ \Rightarrow\ & (s,m) \in \gamma(\check{s}_2, \hat{m}_2) \\ \Rightarrow\ & \gamma(\check{s}_1, \hat{m}_1) \subseteq \gamma(\check{s}_2, \hat{m}_2) \end{aligned}$$

□

In the following, we will extend Γ to sets of abstract states, i. e.

$$\Gamma(\check{s}) = \bigcup\{\Gamma(\hat{s}) \mid \hat{s} \in \check{s}\}$$

To prove the global correctness of the pipeline analysis, we have to specify the abstract versions of the transfer functions $\mathcal{T}_e$, $\hat{\mathcal{T}}_e$. We will define them analogously to the transfer functions in the concrete semantics, which is based on state predicates $\mathcal{F}, \mathcal{N}, \mathcal{H}$ and a retirement function $\mathcal{R}$ together with the state transition function $\mathbb{T}$.

Definition 5.2.3 (Abstract Counterparts): Given the four state predicates $\mathcal{F}$, $\mathcal{N}$, $\mathcal{H}$ and $\mathcal{R}$ according to Definition 4.1.3 and a state transfer function $\mathbb{T}$ from an abstract machine according to Definition 4.1.1. Then predicates $\hat{\mathcal{F}} \subseteq V \times \hat{\mathcal{S}}$, $\hat{\mathcal{N}} \subseteq V \times \hat{\mathcal{S}}$, $\hat{\mathcal{H}} \subseteq \hat{\mathcal{S}}$ and a function $\hat{\mathcal{R}} : V \times \hat{\mathcal{S}} \to \hat{\mathcal{S}}$ are called abstract state predicates for the abstraction described by $\hat{\mathcal{S}}$ and Γ iff the following conditions hold:

1. $\hat{\mathcal{F}}(n, \hat{s}) \Rightarrow \forall s \in \Gamma(\hat{s}) : \mathcal{F}(n, s)$
2. $\neg\hat{\mathcal{F}}(n, \hat{s}) \Rightarrow \forall s \in \Gamma(\hat{s}) : \neg\mathcal{F}(n, s)$
3. $\exists s \in \Gamma(\hat{s}) : \mathcal{N}(n, s) \Rightarrow \hat{\mathcal{N}}(n, \hat{s})$
4. $\hat{\mathcal{H}}(\hat{s}) \Rightarrow \forall s \in \Gamma(\hat{s}) : \mathcal{H}(s)$
5. $\neg\hat{\mathcal{H}}(\hat{s}) \Rightarrow \forall s \in \Gamma(\hat{s}) : \neg\mathcal{H}(s)$
6. If $\hat{\mathcal{R}}(n, \hat{s}) = \hat{t}$ then $\forall s \in \Gamma(\hat{s}) : \mathcal{R}(n, s) \in \Gamma(n, \hat{t})$ and $\forall t \in \Gamma(\hat{t}) : \exists s \in \Gamma(\hat{s}) : t = \mathcal{R}(n, s)$

A function $\hat{\mathbb{T}} : \hat{\mathcal{S}} \to \mathcal{P}(\hat{\mathcal{S}})$ is called abstract transition function iff it satisfies the condition

$$s \in \Gamma(\hat{s}) \Rightarrow \mathbb{T}(s) \in \Gamma(\hat{\mathbb{T}}(\hat{s}))$$

and if for every sequence $\hat{s}_1, \ldots$ with $\hat{s}_{i+1} \in \hat{\mathbb{T}}(\hat{s}_i)$ there exists a $k \in \mathbb{N}$ such that $\hat{\mathcal{F}}(n, \hat{s}_k)$ holds. ■

This definition forces any abstract representation of pipeline states to be exact w.r.t. to the retirement and finishing of instructions. Also, it must always be clear, which instruction is the next one to execute for an abstract state. By using not predicates but maps from $V \times \hat{\mathcal{S}} \to \{0, 1, \top\}$ one could relax these restrictions by using three valued logic functions. This would induce further nondeterminism into the analysis since it would be no longer clear, if and when an instruction has finished execution and where to continue the analysis. For this reason, we choose an exact representation of the state predicates.

With these definition we can define the abstract transfer functions for the global pipeline analysis:

Definition 5.2.4 (Pipeline Transfer Functions): Given an abstract transition function $\hat{\mathbb{T}}$ and abstract state predicates $\hat{\mathcal{F}}, \hat{\mathcal{N}}, \hat{\mathcal{H}}, \hat{\mathcal{R}}$ for the abstract pipeline state domain induced by $\hat{\mathcal{S}}$ and Γ, we define the abstract pipeline transfer functions $\hat{\mathcal{T}}_e : \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z} \to \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z}$ by

$$\hat{\mathcal{T}}_e(\check{s}, \hat{m}) = \bigsqcup\nolimits_{\times} \{\hat{\mathcal{T}}_e^2(\hat{s}, \hat{m}) \mid \hat{s} \in \check{s}\} \tag{5.2.5}$$

Where the function $\hat{\mathcal{T}}_e^2 : \hat{\mathcal{S}} \times \mathcal{Z} \to \mathcal{P}(\hat{\mathcal{S}} \times \mathcal{Z})$ determines for one abstract state/time pair the set of resulting abstract pairs:

$$\hat{\mathcal{T}}^2_{n \to n'}(\hat{s}, \hat{m}) = \begin{cases} \{(\hat{\mathcal{R}}(n, \hat{s}), \hat{m})\} & \text{, if } \hat{\mathcal{F}}(n, \hat{s}) \wedge \hat{\mathcal{N}}(n', \hat{s}) \\ \bigcup\{\hat{\mathcal{T}}^2_{n \to n'}(\hat{t}, \hat{m} + 1) \mid \hat{t} \in \hat{\mathbb{T}}(\hat{s})\} & \text{, if } \neg\hat{\mathcal{F}}(n, \hat{s}) \\ \emptyset & \text{, otherwise} \end{cases} \tag{5.2.6}$$

■

The abstract transfer functions defined this way perform for each abstract state the cycle wise iteration using the abstract transition function $\hat{\mathbb{T}}$ and joining together the resulting sets of abstract states. The upper bound for the execution time is obtained as the maximum of all upper bounds for each iteration of one abstract state.

The $\hat{\mathcal{T}}_e$ are well defined because of the restriction placed on the abstract transition function $\hat{\mathbb{T}}$ in 5.2.3: after finitely many applications of $\hat{\mathbb{T}}$, $\hat{\mathcal{F}}$ holds for the resulting state.

Since we have already shown in Lemma 5.2.2 that the concretization function γ fulfills the necessary premisses of Theorem 3.3.5 it only remains to show that the $\hat{\mathcal{T}}_e$ defined above satisfy condition 3.3.6 to establish that the MOP analysis using the abstract transfer functions 5.2.6 is correct.

This means, we have to show that given abstract state predicates $\hat{\mathcal{F}}, \hat{\mathcal{N}}, \hat{\mathcal{H}}, \hat{\mathcal{R}}$ and an abstract transition function $\hat{\mathbb{T}}$ the following equation holds:

$$\forall S \in \mathcal{P}(\mathcal{S} \times \mathcal{Z}), (\check{s}, \hat{m}) \in \mathcal{P}(\hat{\mathcal{S}}) \times \mathcal{Z} : S \subseteq \gamma(\check{s}, \hat{m}) \Rightarrow \mathcal{T}_e'(S) \subseteq \gamma(\hat{\mathcal{T}}_e(\check{s}, \hat{m})) \quad (5.2.7)$$

Since the $\hat{\mathcal{T}}_e$ are defined as a least upper bound this proof can be simplified by utilizing assumption 3.3.7: by showing that

$$S \subseteq \gamma(\check{s}, \hat{m}) \Rightarrow \mathcal{T}_e'(S) \subseteq \bigcup \{\gamma(\check{t}, \hat{n}) \mid (\check{t}, \hat{n}) \in \hat{S}\} \quad (5.2.8)$$

where $\hat{S} = \{\hat{\mathcal{T}}_e^2(\hat{s}, \hat{m}) \mid \hat{s} \in \check{s}\}$, we can show that 5.2.7 holds. Since the functions $\mathcal{T}_e'$ are defined as a union over the $\mathcal{T}_e$ in 3.2.13, our proof is reduced to showing

$$\begin{aligned} \{(t,n)\} = \mathcal{T}_e(\{(s,m)\}) \wedge (s,m) \in \gamma(\check{s}, \hat{m}) \Rightarrow \\ \exists \hat{s} \in \check{s} : \exists (\check{t}, \hat{n}) \in \hat{\mathcal{T}}_e^2(\hat{s}, \hat{m}) : (t,n) \in \gamma(\check{t}, \hat{n}) \end{aligned} \quad (5.2.9)$$

Note, that the trivial cases of $S = \emptyset$ or $\mathcal{T}_e(S) = \emptyset$ have been omitted from this condition.

Theorem 5.2.10 (Global Correctness): Given abstract state predicates and an abstract transition function, condition (5.2.9) holds, i. e. the pipeline analysis defined by the $\hat{\mathcal{T}}_e$ of 5.2.4 is correct.

Proof:
By inspecting the definition of the $\mathcal{T}_e$ in 4.1.4 we can see that $\{(t,n)\} = \mathcal{T}_{n \to n'}(\{(s,m)\})$ means that $t = \mathcal{R}(\mathbb{T}^k(s))$ and $n = m + k$ for some $k \geq 0$. Therefore, we show by induction on k that the conclusion of 5.2.9 holds.

1. $k = 0$: $t = \mathcal{R}(s) \wedge \mathcal{F}(n,s) \wedge \mathcal{N}(n',s)$ and since $(s,m) \in \gamma(\check{s}, \hat{m})$ there is an $\hat{s} \in \check{s}$ such that $s \in \Gamma(\hat{s})$. By the properties of the abstract state predicates we have $\hat{\mathcal{F}}(n, \hat{s})$ and $\hat{\mathcal{N}}(n', \hat{s})$. Thus $\hat{\mathcal{T}}_{n \to n'}^2(\hat{s}, \hat{m}) = \{(\hat{\mathcal{R}}(n, \hat{s}), \hat{m})\}$ and also $t = \mathcal{R}(n,s) \in \Gamma(\hat{\mathcal{R}}(n, \hat{s}))$. This means that $(\mathcal{R}(n,s), m) \in \gamma(\{\hat{\mathcal{R}}(n, \hat{s})\}, \hat{m})$ holds, proving the claim.
2. $k = k' + 1$: $t = \mathcal{R}(\mathbb{T}^{k'}(\mathbb{T}(s))) \wedge \neg \mathcal{F}(n,s)$, $n = m + k' + 1$. Again, there $\exists \hat{s} \in \check{s}$ such that $s \in \Gamma(\hat{s})$. We also have $\neg \hat{\mathcal{F}}(n, \hat{s})$ from the definition of the abstract state predicates. By the definition of the abstract transition function $\hat{\mathbb{T}}$ we have that $(\mathbb{T}(s), m+1) \in \gamma(\hat{\mathbb{T}}(\hat{s}), \hat{m}+1)$. Applying the induction hypothesis to $\{(t,n)\} = \mathcal{R}(\mathbb{T}^{k'}(\mathbb{T}(s), m+1))$ we have

that there exists a $\hat{u} \in \hat{\mathbb{T}}(\hat{s}) : \exists(\check{t},\hat{o}) \in \hat{\mathcal{T}}^2_{n\to n'}(\hat{u},\hat{m}+1) : (t,n) \in \gamma(\check{t},\hat{o})$. By the definition of $\hat{\mathcal{T}}_e$ for the case $\neg\hat{\mathcal{F}}(n,\hat{s})$ we have that $(\check{t},\hat{o}) \in \hat{\mathcal{T}}_{n\to n'}(\hat{s},\hat{m})$, proving the claim.

Thus, we can conclude that the abstract pipeline transfer functions $\hat{\mathcal{T}}_e$ define a correct MOP analysis, if abstract pipeline predicates $\hat{\mathcal{F}}, \hat{\mathcal{N}}, \hat{\mathcal{H}}, \hat{\mathcal{R}}$ and an abstract transition function $\hat{\mathbb{T}}$ are given.

□

Note that there are no conditions on the concretization function Γ, mapping one abstract state to a set of concrete states. We even do not require that the set $\hat{\mathcal{S}}$ is a domain or even has an ordering defined on it. This makes it easier to define new abstractions with different $\hat{\mathcal{S}}$ and Γ.

In the following, we will present an abstraction that corresponds to our approach of formulating $\mathbb{T}$ with unit updates using delayed and instantaneous signals.

5.3 The Abstraction Using Unit Updates

As in Chapter 4 we will use the same units communicating via delayed or instantaneous signals. Abstract unit updates use abstract domains for the types of the unit components (but the components stay the same). An abstract state is then obtained as the union of all the (abstract) units. In the same way we obtained a concrete state as the union of the (concrete) units in the previous chapter.

The abstract domains are connected to the concrete domains of the unit components via the concretization Γ, which will be defined for every component below. The result of applying Γ to an abstract state $\hat{s}$ is then composed from the results of applying it to the components.

As discussed earlier in Section 4.2.2, using abstractions for the components may prohibit the precise represention of the components values. Since the components values are used in the evaluation of expressions, information loss in the components will introduce nondeterminism in the evaluation of expressions in the unit update programs. Nondeterminism in expressions means that also the execution of instructions of the unit update program is nondeterministic. Thus, we may have several successor states for one state after a unit update.

Because only the domains of the components in the units are changed, units and thus abstract states are still represented as environments. The difference between (concrete) environments and abstract ones is that abstract environments map the components names to abstract values from the abstract domains. For components that do not contain abstract types, the abstract domains are the same as the concrete ones: the information in these components is represented exactly in the analysis. For abstract types, we use new underlying domains whose elements are approximations to the elements of the domains of the abstract types in the concrete semantics. That is, one element from the abstract domain represents a set of elements in the concrete domain, in the same way as an abstract state represents a set of concrete states.

We will denote abstract environments by $\hat{\rho}$, abstract delayed signals by $\hat{\delta}$ or ρ^{δ}. In the concrete case, we used a function P to associate a domain of concrete values, $P(\mathrm{id})$ to an abstract type $\mathsf{abstr}(\mathrm{id})$, cf. Section 4.2.3. Now we will use a function $\hat{P}$ to associate new (abstract) value domains $\hat{P}(\mathrm{id})$ to the abstract types. For every such pair $P(\mathrm{id})$ and $\hat{P}(id)$ we assume that there is a *concretization* function that maps an element from $\hat{P}(\mathrm{id})$ to a set of values in $P(\mathrm{id})$ that are approximated by that element. We will call this function Γ, too, like the concretization function for abstract states. Thus, for every abstract type $\mathsf{abstr}(\mathrm{id})$, $\Gamma : \hat{P}(\mathrm{id}) \to \mathcal{P}(P(\mathrm{id}))$.

The concretization function Γ, mapping an abstract state to a set of concrete states represented by it, can then be defined by applying Γ to the values of the environmental mapping $\hat{\rho}$, which is the abstract counterpart of ρ, and subsequently to mappings $\hat{\mu}$ and $\hat{\nu}$, the counterparts of μ and ν. We also extend Γ to all domains, defined by our type system:

$$\begin{array}{lcl}
\Gamma(\hat{\rho}) & = & \{\rho \mid l \mapsto \hat{v} \in \hat{\rho} \wedge l \mapsto v \in \rho \wedge v \in \Gamma(\hat{v})\} \\
\Gamma((\hat{v}_1,\ldots,\hat{v}_n)) & = & \{(v_1,\ldots,v_n) \mid v_i \in \Gamma(\hat{v}_i)\} \\
\Gamma((\mathrm{id},\hat{v})) & = & \{(\mathrm{id},v) \mid v \in \Gamma(\hat{v})\} \\
\Gamma(z) & = & \{z\} \\
\Gamma(b) & = & \{b\}
\end{array} \tag{5.3.1}$$

Structures and arrays are already covered by the case for abstract environments. Now that we have defined the abstract states completely as abstract environments, we continue by giving the abstract counterparts of the transition functions $\mathbb{T}$, $\mathbb{T}_i^S$ and $\mathbb{T}_i^u$. The abstract semantics of a unit update, $\hat{\mathbb{T}}_i^u$ will be given by an abstract version $\rightrightarrows_p$ of the relation $\rightarrow_p$. This will also be defined by a set of inference rules.

In the concrete update we had the mapping $\delta_\perp$, representing the unasserted state of delayed signals. We will make use of its abstract counterpart, $\hat{\delta}_\perp$. Here, $\hat{\delta}_\perp$ can be chosen from a variety of abstract values. However, the one chosen must be an approximation to $\delta_\perp$, i. e. $\delta_\perp \in \Gamma(\hat{\delta}_\perp)$ must hold. Also, there is an abstract counterpart for the state $\perp$, $\hat{\perp}$ with $\perp \in \Gamma(\hat{\perp})$.

Definition 5.3.2 (Abstract Transition Function): The function

$$\hat{\mathbb{T}}(\hat{\mu}) = \begin{cases} \{\hat{\perp}\} & \text{, if } \hat{\mathcal{H}}(\hat{\mu}) \\ \oplus(\bigcup\{\hat{\mathbb{T}}_2^S(\{1,\ldots,N\},\hat{\nu}',\hat{\delta}') \mid \\ \quad (\hat{\nu}',\hat{\delta}') \in \hat{\mathbb{T}}_1^S(\{1,\ldots,N\},\hat{\mu},\hat{\delta}_\perp)\}) & \text{, otherwise} \end{cases}$$

(where $\oplus(\hat{S}) = \{(\hat{\nu}' \oplus \hat{\delta}') \downarrow (\mathrm{Lab}^c \cup \mathrm{Lab}^\delta) \mid (\hat{\nu}', \hat{\delta}') \in \hat{S}\}$) is called the *abstract transition function for unit updates*. ■

The functions $\hat{\mathbb{T}}^S_{1,2}$ traverse all units according to the dependencies between units for half-cycle 1 and 2:

Definition 5.3.3 (Unit Traversion): The functions $\hat{\mathbb{T}}^S_{1,2} : \mathcal{P}(\{1,\ldots,N\}) \times [\hat{\nu}] \times [\hat{\delta}] \to \mathcal{P}([\hat{\nu}] \times [\hat{\delta}])$ are defined by

$$\hat{\mathbb{T}}^S_i(U, \hat{\nu}, \hat{\delta}) = \begin{cases} \{(\hat{\nu}, \hat{\delta})\} & \text{, if } U = \emptyset \\ \bigcup\{\hat{\mathbb{T}}^S_i(U \backslash \{u\}, \hat{\nu} \oplus \hat{\nu}', \hat{\delta} \oplus \hat{\delta}') \mid & \\ \quad (\hat{\nu}', \hat{\delta}') \in \hat{\mathbb{T}}^u_i(\hat{\nu})\} & \text{, otherwise} \\ \quad \text{where } u = \min_{\prec_{\mathbb{D}}}(U') & \end{cases}$$

■

The functions $\hat{\mathbb{T}}^u_i$ perform a half-cycle update for unit u:

Definition 5.3.4 (Abstract Unit Update): The functions $\hat{\mathbb{T}}^u_i : [\hat{\nu}] \to \mathcal{P}([\hat{\nu}] \times [\hat{\delta}])$ are defined by

$$\hat{\mathbb{T}}^u_i(\hat{\nu}) = \{(\hat{\rho} \downarrow (\mathrm{Lab}^c \cup \mathrm{Lab}^\delta \cup \mathrm{Lab}^\iota), \hat{\delta}) \mid p \rightrightarrows_p^{(\hat{\nu}, \hat{\delta}_\perp)} (\hat{\rho}, \hat{\delta})\}$$

where $p = p_{u_i}$ is the program for unit u in half-cycle i. ■

These three functions are defined similarly to their concrete counterparts; the only significant change is that one abstract state update may produce a set of possible successor states.

As in the case of the concrete semantics, we use a set of inference rules to define the relation $p \rightrightarrows_p^{\hat{\tau}} \hat{\tau}'$ mapping abstract environments to pairs of new abstract environment and newly emitted abstract delayed signals.

As we can no longer expect that the abstract values can represent information exactly, the relation $\rightrightarrows_p$ is no longer deterministic, i. e. there may be several $(\hat{\rho}, \hat{\delta})$ for one $(\hat{\nu}, \hat{\delta}_\perp)$ with $p \rightrightarrows_p^{(\hat{\nu}, \hat{\delta}_\perp)} (\hat{\rho}, \hat{\delta})$. In our framework, this nondeterminism comes solely from the evaluation of expressions using abstract values instead of the concrete ones: the inference rules for expression evaluation are nondeterministic. The inference rules for program evaluation are the same as in the concrete case.

Taking the semantics from Section 4.2.3, we only have to replace all predefined functions by abstract counterparts. Predefined functions are the *only* source of nondeterminism in the abstract semantics. We replace the meaning of a predefined function $f = [\![p]\!]$, $f : D_1 \times \cdots \times D_n \to D \times [\rho]$ by an abstract counterpart

$\hat{f} = \widehat{[\![p]\!]}$, $\hat{f} : \hat{D}_1 \times \cdots \times \hat{D}_n \to \mathcal{P}(\hat{D} \times [\hat{\rho}])$. That is, the argument domains of f are replaced by their abstract counterparts and the result domain is a finite *set* of abstract values, coming from the abstract counterpart of the concrete result domain. Here, the result is a pair of value, environment in the concrete case and a set of pairs of abstract value, abstract environment in the abstract case. By this, we can model the fact that a certain computation may not have a uniquely determined result in the analysis and that the result of a function can be undetermined, but be within a set of values.

Thus, we can define the evaluation of an expression e under an abstract environment $\hat{\rho}$, denoted by $e \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{v}, \hat{\rho}')$ by nearly the same set of inference rules, as in the concrete case:

(50) $\dfrac{}{n \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (n, \hat{\rho})}$, if n is an integer

(51) $\dfrac{}{b \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (b, \hat{\rho})}$, $b \in \{\mathsf{true}, \mathsf{false}\}$

(52) $\dfrac{e_1 \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{v}_1, \hat{\rho}_1), \ldots, e_n \rightrightarrows^{\hat{\rho}_{n-1}}_{\hat{\mathcal{E}}} (\hat{v}_n, \hat{\rho}_n)}{(e_1, \ldots, e_n) \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} ((\hat{v}_1, \ldots, \hat{v}_n), \hat{\rho}_n)}$

(53) $\dfrac{}{\mathrm{id} \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{\rho}(\mathrm{id}), \hat{\rho})}$, if $\mathrm{id} \in \mathrm{dom}(\hat{\rho})$

(54) $\dfrac{}{\mathrm{id} \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} ((\mathrm{id}, ()), \hat{\rho})}$, if id is an enum identifier

(55) $\dfrac{e \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{v}, \hat{\rho}')}{\mathrm{id}\ e \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} ((\mathrm{id}, \hat{v}), \hat{\rho}')}$, if id is an enum identifier

(56) $\dfrac{e \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{v}, \hat{\rho}')}{\mathrm{id}[e] \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{\rho}'(\mathrm{id})(\hat{v}), \hat{\rho}')}$

(57) $\dfrac{n \rightrightarrows^{\hat{\rho} \oplus \hat{\rho}(\mathrm{id})}_{\hat{\mathcal{E}}} (\hat{v}, \hat{\rho}')}{\mathrm{id}.n \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\hat{v}, \hat{\rho}' \uparrow \mathrm{dom}(\hat{\rho}(\mathrm{id})) \oplus (\hat{\rho} \downarrow \mathrm{dom}(\hat{\rho}(\mathrm{id}))))}$

(58) $\dfrac{}{\texttt{Received}\,\mathrm{id}(\mathrm{id}_1, \ldots, \mathrm{id}_n) \rightrightarrows^{\hat{\rho}}_{\hat{\mathcal{E}}} (\mathsf{true}, \hat{\rho} \oplus \{\mathrm{id}_i \mapsto \hat{v}_i \mid 1 \leq i \leq n\})}$, if

$\mathrm{id} \mapsto (\hat{v}_1, \ldots, \hat{v}_n) \in \hat{\rho} \wedge \mathrm{id} \in \mathrm{Lab}^{\iota}$ or
$\mathrm{id} \in \mathrm{Lab}^{\delta} \wedge \hat{\rho}(\mathrm{id}) \neq \hat{\delta}_{\perp}(\mathrm{id})$

(59) $$\frac{}{\texttt{Received}\ \mathrm{id}(\mathrm{id}_1,\ldots,\mathrm{id}_n) \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}} (\mathsf{false}, \hat{\rho})}\text{, if}$$

$$\begin{array}{l}\mathrm{id} \in \mathrm{Lab}^{\iota} \wedge \nexists \mathrm{id} \mapsto \hat{v} \in \hat{\rho} \text{ or}\\ \mathrm{id} \in \mathrm{Lab}^{\delta} \wedge \hat{\rho}(\mathrm{id}) = \hat{\delta}_{\perp}(\mathrm{id})\end{array}$$

(60) $$\frac{e_1 \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}} (\hat{v}_1, \hat{\rho}_1), \ldots, e_n \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}_{n-1}} (\hat{v}_n, \hat{\rho}_n)}{p(e_1,\ldots,e_n) \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}} (\hat{v}, \hat{\rho}')}\text{, if } (\hat{v}, \hat{\rho}') \in \widehat{[\![p]\!]}(\hat{\rho}_n, \hat{v}_1, \ldots, \hat{v}_n)$$

Here, the nondeterminism is introduced by rule (60), where one of the return value/environment pairs of the abstract function is chosen. Naturally, we have to impose a condition on the abstract version of a predefined function, relating it to its concrete counterpart. This condition is, not surprisingly, that every concrete computation by a predefined function must be in the concretization of the abstract computation, if the concrete arguments are represented by the abstract ones:

$$\begin{array}{l}(\rho, v_1, \ldots, v_n) \in \Gamma((\hat{\rho}, \hat{v}_1, \ldots, \hat{v}_n)) \Rightarrow\\ [\![p]\!](\rho, v_1, \ldots, v_n) \in \bigcup\{\Gamma((\hat{v}, \hat{\rho}')) \mid (\hat{v}, \hat{\rho}') \in \widehat{[\![p]\!]}(\hat{\rho}, \hat{v}_1, \ldots, \hat{v}_n)\}\end{array} \tag{5.3.5}$$

The rules for $\rightrightarrows_p$ are the same as those for $\rightarrow_p$, just replacing $\rightarrow_{\mathcal{E}}$ by $\rightrightarrows_{\hat{\mathcal{E}}}$.

This abstract inference system has the property that any concrete computation has an abstract one starting from correct approximations:

Theorem 5.3.6 (Abstract Computation Correctness): Let a concrete derivation $p \rightarrow_p^{(\nu, \delta_\perp)} (\rho, \rho^\delta)$ be given and $(\nu, \delta_\perp) \in \Gamma((\hat{\nu}, \hat{\delta}_\perp))$, then there exist $\hat{\rho}$ and $\hat{\rho}^\delta$ with $p \rightrightarrows_p^{(\hat{\nu}, \hat{\delta}_\perp)} (\hat{\rho}, \hat{\rho}^\delta)$ and $(\rho, \rho^\delta) \in \Gamma((\hat{\rho}, \hat{\rho}^\delta))$.

Proof:
As the only difference between the set of inference rules is the relation used to denote evaluation of expressions, it is sufficient to prove that the concrete evaluation of an expression is approximated by the abstract evaluation, i. e. we have to show

$$e \rightarrow_{\mathcal{E}}^{\rho} (v, \rho') \wedge \rho \in \Gamma(\hat{\rho}) \Rightarrow \exists \hat{v}, \hat{\rho}' : e \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}} (\hat{v}, \hat{\rho}') \wedge (v, \rho') \in \Gamma((\hat{v}, \hat{\rho}')) \tag{5.3.7}$$

Here, the only rule different between the concrete and abstract computations is the evaluation of predefined functions. For them, we have to show that there exist $\hat{\rho}', \hat{v}$ such that $p(e_1, \ldots, e_n) \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}} (\hat{v}, \hat{\rho}')$. We can assume that there exists $\hat{v}_i$ and $\hat{\rho}_i$, $1 \leq i \leq n$, such that $e_i \rightrightarrows_{\hat{\mathcal{E}}}^{\hat{\rho}_{i-1}} (\hat{v}_i, \hat{\rho}_i)$ (by induction on the length of the concrete inference tree). For those we also have $(v_i, \rho_i) \in \Gamma((\hat{v}_i, \hat{\rho}_i))$. By the condition on the abstract versions of the predefined function meaning, we have $(v, \rho') = [\![p]\!](\rho, v_1, \ldots, v_n) \in \bigcup\{\Gamma((\hat{v}, \hat{\rho}')) \mid$

$(\hat{v},\hat{\rho}') \in \widehat{[\![p]\!]}(\hat{\rho},\hat{v}_1,\dots,\hat{v}_n)\}$. Thus, there exist $(\hat{v},\hat{\rho}') \in \widehat{[\![p]\!]}(\hat{\rho},\hat{v}_1,\dots,\hat{v}_n)\}$ such that $(v,\rho') \in \Gamma((\hat{v},\hat{\rho}'))$, proving the claim.

□

This theorem in turn implies that for $\nu \in \Gamma(\hat{\nu})$ we have $\mathbb{T}_i^u(\nu) \in \Gamma(\hat{\mathbb{T}}_i^u(\hat{\nu}))$ by the definitions of $\mathbb{T}_i^u$ and $\hat{\mathbb{T}}_i^u$. This can then be used to prove the

Theorem 5.3.8 (Abstract Unit Traversion Correctness): If it is true for a pair (ν,δ') that $(\nu,\delta') \in \Gamma((\hat{\nu},\hat{\delta}'))$ then we also have that

$$\mathbb{T}_i^S(U,\nu,\delta') \in \Gamma(\hat{\mathbb{T}}_i^S(U,\hat{\nu},\hat{\delta}'))$$

Proof:
We do induction on the number of element in the set U.

- $|U| = 0$: Then $\mathbb{T}_i^S(U,\nu,\delta') = (\nu,\delta')$ and $\hat{\mathbb{T}}_i^S(U,\hat{\nu},\hat{\delta}') = \{(\hat{\nu},\hat{\delta}')\}$. The claim trivially holds.
- Let $|U| = n+1, n \geq 0$. The assumptions allow us to conclude that $\mathbb{T}_i^u(\nu) = (\nu'',\delta'') \in \Gamma(\hat{\mathbb{T}}_i^u(\hat{\nu}))$. This means there exist $(\hat{\nu}'',\hat{\delta}'') \in \hat{\mathbb{T}}_i^u(\hat{\nu})$ with $(\nu'',\delta'') \in \Gamma((\hat{\nu}'',\hat{\delta}''))$. This also means that
  - $\nu \oplus \nu'' \in \Gamma(\hat{\nu} \oplus \hat{\nu}'')$
  - $\delta' \oplus \delta'' \in \Gamma(\hat{\delta}' \oplus \hat{\delta}'')$

  By using the induction hypothesis we have that $\mathbb{T}_i^S(U \setminus \{u\},\nu \oplus \nu'',\delta' \oplus \delta'') \in \Gamma(\hat{\mathbb{T}}_i^S(U \setminus \{u\},\hat{\nu} \oplus \hat{\nu}'',\hat{\delta}' \oplus \hat{\delta}'')$ holds, which proves our claim by the definition of $\mathbb{T}_i^S$ and $\hat{\mathbb{T}}_i^S$

□

Using this theorem we can finally prove that our $\hat{\mathbb{T}}$ is a correct abstract transition function.

Theorem 5.3.9 (Abstract Transition Function Correctness): The function $\hat{\mathbb{T}}$ defined in (5.3.2) is an abstract transition function as defined in (5.2.3).

Proof:
First, $\hat{\mathbb{T}}$ always reaches a state in which $\hat{\mathcal{F}}$ holds, because $\hat{\mathcal{F}}$ correctly approximates $\mathcal{F}$ and that one is guaranteed to finish every instruction eventually. Then, we have to show that $\hat{\mathbb{T}}$ is a correct approximation to $\mathbb{T}$, i. e.

$$\mu \in \Gamma(\hat{\mu}) \Rightarrow \mathbb{T}(\mu) \in \Gamma(\hat{\mathbb{T}}(\hat{\mu}))$$

Assume that $\mathcal{H}(\mu)$ is true. Then $\mathbb{T}(\mu) = \bot$ and because of the properties imposed on $\hat{\mathcal{H}}$ we have that $\hat{\mathbb{T}}(\hat{\mu}) = \{\hat{\bot}\}$, where $\bot \in \Gamma(\hat{\bot})$ is true. If

$\neg\mathcal{H}(\mu)$ then also $\neg\hat{\mathcal{H}}(\mu)$. By the previous theorem we have that $(\nu_1,\delta_1) = \mathbb{T}_1^S(M,\mu,\delta_\perp) \in \Gamma(\hat{\mathbb{T}}_1^S(M,\hat{\mu},\hat{\delta}_\perp))$, where $M = \{1,\dots,N\}$. This means there exist $(\hat{\nu}_1,\hat{\delta}_1) \in \hat{\mathbb{T}}_1^S(M,\hat{\mu},\hat{\delta}_\perp)$ with $(\nu_1,\delta_1) \in \Gamma((\hat{\nu}_1,\hat{\delta}_1))$. Applying that theorem once more we obtain $(\nu_2,\delta_2) = \mathbb{T}_2^S(M,\nu_1,\delta_1) \in \Gamma(\hat{\mathbb{T}}_2^S(M,\hat{\nu}_1,\hat{\delta}_1))$. By the properties of the $\oplus$ operator we then have

$$\mathcal{M}(\nu_2,\delta_2) \in \Gamma(\oplus(\bigcup\{\hat{\mathbb{T}}_2^S(M,\hat{\nu}',\hat{\delta}') \mid (\hat{\nu}',\hat{\delta}') \in \hat{\mathbb{T}}_1^S(M,\hat{\mu},\hat{\delta}_\perp)\})$$

which proves the claim.

□

This concludes the correctness proof for our unit update style pipeline analysis. The following sections will give details on the analyses for the two example models presented in the previous chapter.

5.3.1 Analysis for the MCF 5307

For this analysis two major abstractions were made. First, the cache is abstracted by a one-way direct mapped abstract cache with 128 ways. All updates on the cache are replaced by the abstract must cache update function for this cache, [Fer97]. All functions that classify a cache access are replaced by the classification function for this cache, more precisely

$$\mathrm{cl}(m,c) = \begin{cases} \{\mathbf{hit}\} & \text{, if } m \in c(\mathrm{set}(m)) \\ \{\mathbf{hit},\mathbf{miss}\} & \text{, otherwise} \end{cases}$$

$\mathrm{cl}(m,c)$ returns the possible concrete values for the classification of memory block m with the abstract cache c.

The other abstraction was more far reaching: all addresses that may not be known precisely in components or signals were replaced by intervals of addresses. This effects the addresses of data accesses in the execution pipeline, $\mathit{read}(a)$ and $\mathit{write}(a)$ and the state in the EX and bus unit. The predefined functions are replaced by abstract versions that operate on intervals. As an example, the cache update with such an interval has to touch all cache lines covered by the interval, while no block can be placed in the cache if the interval is not precise (one element). This results in loss of information about the (unified) cache for any cached imprecise data access. The other important abstraction is the function that returns the timings for a memory area referenced by an address. This function must now return a set of possible timings, which are all the timings for the memory area spanned by the interval. The other necessary abstractions are obvious from the concrete model.

The abstract state predicates are the same as the concrete ones, i. e. we represent this information exactly.

The other abstract (implicit) components in the concrete model (memory, registers) are abstracted by the domain with just the element $\top$, i. e. no knowledge is available for these components.

5.3.2 Analysis for the PPC 755

Two major abstractions have been introduced: caches and memory address intervals. The caches are the abstractions of the PowerPC 755 caches, which only represent must information: some memory blocks can be guaranteed to be in the cache. Thus, the abstract cache classification function is the same as for the ColdFire. The cache update function is the one modeling the PowerPC 755 abstract cache, which is able to obtain at most information for half of the blocks in a set.

All data address components and signals are abstracted by address intervals. This is relevant for the LSU and the store buffer as well as the bus unit and the chip set unit, CSU. As for the ColdFire, all functions working on such addresses were changed to abstract versions that return the collection of information valid for the interval, e. g. access timings.

The function that gives the number of cycles for instructions to execute in the functional units is abstracted by a version that always returns the set of possible cycle count values, e. g. $\{1,2,3\}$ for the IU if a multiplication is performed. Functions that determine if a speculative branch has been resolved correctly are abstracted by a version that uses infeasibility information from the value analysis, cf. Section 6.2, to sometimes limit the set of possible return values. If this is not possible, it returns the set $\{\mathsf{false}, \mathsf{true}\}$.

The abstract state predicates are the same as the concrete ones.

Here too, memory and registers are abstracted by the single element domain $\top$ as for the ColdFire analysis.

5.4 Nondeterminism

The abstract computation semantics $\rightrightarrows_p$ of the pipeline models is nondeterministic, because some decisions are based on data, for which no precise information (or no information at all) is available due to the abstraction applied on the components. The abstract transition function collects all possible results from the computation semantics. The correctness of the analysis has been shown correct for this transition function.

Naturally, it is costly to collect all possible results from the nondeterministic computation semantics. If we are primarily interested in the WCET of a program we only need to consider the result(s) for $\rightrightarrows_p$ that can lead to the WCET. For architectures without timing anomalies this would be the local worst-case result, e. g. the state resulting in a cache miss if a cache access could not be precisely classified as hit or miss in a cycle update. In this case, we can define a *deterministic* abstract computation semantics $\rightrightarrows_p{}^l$ which is defined by

$$p \overset{l}{\rightrightarrows}{}^{\hat{\tau}}_{p} \tau' \iff p \rightrightarrows^{\hat{\tau}}_{p} \tau' \wedge \tau' \text{ represents local worst-case}$$

Even for architectures with timing anomalies, we can make $\rightrightarrows_p$ "less nondeterministic" by only considering result states τ' that may lead to the global worst-case, leaving out those states that are *guaranteed* to never participate in a global worst-case execution time scenario. Unfortunately, it is not easy to decide which local case cannot result in a global worst-case. For such a proof, it must be shown that all state traces starting from this state are not longer than all possible traces starting from other states. As there are subtle interdependencies between the components of the pipeline model we have not introduced such a reduction of nondeterminism and stayed on the safe side of looking at all possible results of $\rightrightarrows_p$ in our analysis.

This complete exploration of the state space seems to be prohibitively expensive in terms of memory usage and computation time. Fortunately, the problem is reduced by the following observations:

- The number of possible result states is bounded as there are not many nondeterministic choices to be made in one cycle. Additionally, if a cycle update contains nondeterministic choices with multiple successor states, the next one will probably be deterministic for each of them, so the state graph does not branch at every level (cycle).

- The analysis itself records the number of executed cycles not with every state but rather has an upper bound for all cycles asociated with a set of abstract states. Thus, states differing only in the number of cycles simulated on them fall together, reducing the number of states in the set. In fact, it can be seen in the actual outputs of the analysis that a lot of states fall together this way, reducing the state space further.

- We can always apply a *widening* to a set of states. This widening computes a smaller set of states, which is still a correct approximation for the original state set. In practice, we use this technique to reduce sets of states that differ only in their abstract cache contents, by replacing them with a single state, whose cache content is the least upper bound (in the abstract cache domain) of all the abstract caches of the original states.

In practice there are a number of significant sources of nondeterminism.

- Cache accesses that cannot be classified precisely,
- external memory accesses whose address is not known precisely enough,
- varying execution times of instructions.

The nondeterminism due to caching comes from the fact that the cache analyses are not able to classify every access precisely as cache hit or miss. These accesses must then be treated as cache miss and as cache hit, resulting in two successor states. The problem is even more present for the caches of the ColdFire 5307 and PowerPC 755, because the cache analysis is never able to classify accesses as guranteed cache misses due to the missing cache may analysis. Also, the must analysis for the caches can only predict a reduced portion of the cache contents (at most 1/4 for ColdFire and 1/2 for PPC 755). Because instruction cache accesses can cross cache line boundaries, up to two cache lines may be accessed for a single fetch. Thus, this can result in up to four successor states if these accesses cannot be classified exactly by the analysis.

Varying execution times for functional units due to shortcut evaluation in the units (e. g. multiplication of "short" numbers with leading zero bits) results in as many successor states for one state as there are possible execution times for the functional units. It may be possible to reduce this nondeterminism by exporting more information from the value analysis (cf. Chapter 6) about the values of the operands. However, this makes the analysis itself more complicated. The effects of this nondeterminism are limited to a few cycles most of the time because the resulting states fall together after the instruction has completed and the processor waits for main memory to conclude an instruction fetch or data access.

The most severe source of nondeterminism arises from memory accesses. The configuration of a real-time system typically contains a significant number of memory areas with different access characteristics and access times. If the address of a data access is not known precisely, all memory areas that intersect with the interval used as abstraction of the data access address must be considered as possible targets for the access. This results in as many successor states as there are memory areas configured in the system. This is especially problematic for analyses of functions in the program that contain arguments used as indices into an array. As the value of these arguments are unknown initially, all array accesses are assumed to span the whole address range. The result is typically a large number of states and a loss of precision for the WCET result because slow memory areas (controller register maps, etc) are probably never really the target of these accesses during a concrete execution of the function. This can also lead to a chain of branches in the state graph. Some instructions access memory multiple times to save and restore register sets at function entry or exit. If for every access this branching in the state graph occurs then a lot of useless states are generated. One can circumvent this problem by the assumption that those instructions will access memory only in the same memory area during execution. Then, a state component is added that records the memory area that is currently being accessed. This component is initialized on the first access of the instruction to a different value in the successor states according to the memory areas possibly targeted by the instruction. This way the state graph branches only once for the first memory access of the instruction.

In practice, many of the nondeterministic choices can be avoided by introducing *user annotations* into the analysis. These are assertions done by the user of the analysis that add knowledge the analysis cannot infer itself. E.g. the range of a function argument may be restricted to a certain interval due to knowledge on the size of the arrays accessed via this argument. This will drastically reduce the branching due to unknown memory accesses and improve the precision of the results. As mentioned in Chapter 6, there are a lot of annotations that can be added to the analysis this way to improve the results and decrease the costs of the analysis itself. There has even been implemented a mode that makes the computation deterministic by only using the local worst-case in pipeline state computations.

5.5 Parallelism in a Sequential Data-Flow Analysis

The concept of a control-flow graph relies on the *sequential* execution of the instructions in the program. Thus, also data-flow analysis assumes that instructions are executed sequentially and extensions to parallel programs have not been too successful. This seems to make the analysis of pipelines by data-flow analysis impossible, since pipelines are solely there to parallelize program execution as much as possible.

Luckily the parallelism in a machine program is of a limited form, namely designed in such a way as to guarantee the sequentiality of the *effects* of the program on the architectural state. This is mostly due to the fact that certain kinds of *exceptions* in the system must be guaranteed to be *precise*. Consider for example an operating system with virtual memory. If a page fault occurs because the accessed page has not yet been mapped in from backing store an exception is raised. The handler routine must be able to continue the execution of the program with the instruction that caused the fault after it has mapped in the page. For this, the address of that instruction must be known and all effects of instructions before this instruction must be known to be commited to the architectural state.

This does not keep the architectures from dropping instructions that have no effect as soon as possible or to perform actions that have no effects on the state

(e. g. speculative data accesses). The important consequence of all these considerations is that there exists a unique criterion, when an instruction has left the pipeline: its effects have been commited. We can use this to enforce a virtual sequentiality for our analyses as expressed by the notion of state predicates in Definition 4.1.3.

In reality, the fact that certain instructions may be dropped without executing in the processor (noops, folded branches) enforces additional components in the state modeling that record (for the analysis) that these instructions have been encountered. By careful bookkeeping of these instructions, we can use the sequential data-flow analysis to analyze the parallel pipeline of a processor.

One example of such bookkeeping comes from the fact that an instruction stream is prefetched long before it even starts execution. Intuitively, this means we have a look-ahead along edges in the CFG for the instructions fetched. We must record not only the addresses of these instructions but also the *contexts* under which they occur in the CFG, i. e. a look-ahead in the supergraph formed by unrolling the CFG with contexts. For this reason, the analyses have state components that record the context number for prefetched instructions. These can be viewed as the information needed to compute the state predicate for the next instruction.

5.6 Other Caveats

More practical things to consider are mainly related to the efficiency of the analysis. The pipeline updates need a lot of information about the instructions currently in the pipeline. For example, the registers written to by an instruction (which is identified by its address in the program code). Recomputing this information in the update program for every cycle where it is accessed is costly. As this information is fixed for the program it can be precomputed. For this, a single pass along the CFG is done initially and all relevant information is collected in a hash table indexed by the instruction addresses.

Chapter 6

A WCET Toolframe

No practical analysis for out-of-order superscalar processors has been presented. Due to the problems of very complex behavior, timing anomalies, and the problem of finding all relevant processor features, we do not think that a practical, safe, and tight analysis is feasible.

[Eng02]

Start by doing what is necessary, then what is possible, and suddenly you are doing the impossible

St. Francis of Assisi

The novel approach presented in this thesis explains how a systems hardware can be modeled and how a correct (pipeline) analysis of the system's timing behavior can be derived from this model. However, a real-life tool has many more components aside from the pipeline analysis. The whole framework is the result of several PhD theses and has been developed by the company AbsInt.

Figure 6.1 shows the toolchain that makes up AbsInt's WCET analysis tool, **aiT**. Eight main phases, realized by separate programs, can be identified:

1. The *reconstruction of the control-flow graph* takes a binary executable and extracts the CFG of the part to be analyzed into the intermediate representation, CRL. This phase is discussed in more detail in Section 6.1.

2. The second phase, *loop transformation*, searches for loops in the CFG extracted by the first phase and marks them. As explained in Section 3.3.1, loops are extracted in the intermediate representation as virtual procedures

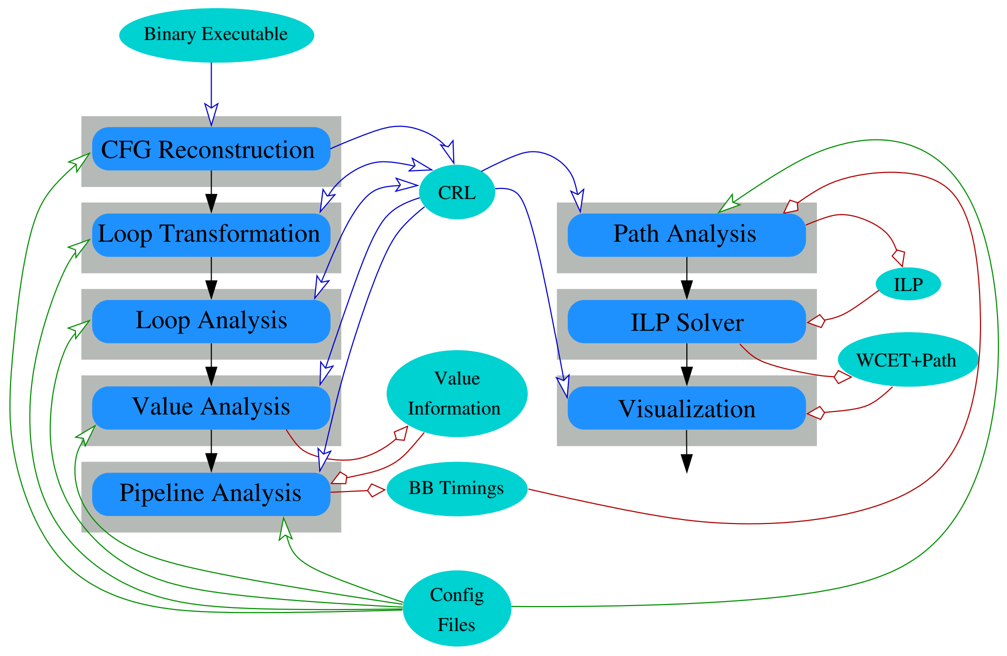


Figure 6.1: The **aiT** toolchain

so that the common context separation techniques can be applied for the precise analysis of loops.

3. The *loop analysis* phase tries to find the iteration bounds for the loops in the program statically. This is done by a data-flow analysis that derives the loop bounds for certain code patterns based on safe information. E.g. if a variable is incremented by a constant increment upon every iteration and the loop termination condition is a comparison against a fixed number, the analysis will validate these conditions and derive the iteration bound from the initial value of the loop iteration variable, the increment and the comparison value. Many loop bounds can be automatically detected by this analysis. Thus, only a small number of loop bounds must be given manually by the user of the WCET tool.

4. The next phase, *value analysis*, uses a data-flow analysis to determine intervals for the data memory accesses performed by the program. There intervals present safe bounds for the addresses of these accesses computed

dynamically at run-time. Section 6.2 has more details about this important phase. The value analysis writes its results into a separate result file, which is read by the next phase.

5. The *pipeline analysis* implements the concepts introduced in this thesis and determines WCET bounds for the basic blocks of the program. These results are written out into a separate result file, which is read by the path analysis phase.

6. The *path analysis* generates an *integer linear program* (ILP) from the CFG of the program and the results of the pipeline analysis. The objective function of the ILP is the execution time of the program, which is then maximized. This phase can be influenced by various configuration statements to exclude special paths or whole code pieces and to add extra constraints to model special relations between parts of the program. This phase is the work of Henrik Theiling and is described in detail in his PhD thesis [The02].

7. The seventh phase solves the ILP produced by the path analysis with the help of an ILP solver. In addition, this phase translates the results of the ILP solver back to a WCET for the whole program and constructs a *worst-case path* through the CFG of the program. This path is one of the (possibly) many paths through the CFG upon which the computed WCET for the whole program would occur. Also, this part computes WCET contributions for all the basic blocks and functions in the program[1].

8. The last phase computes and displays a graphical and interactively explorable representation of the WCET results. Section 6.4 has more details and examples about this phase.

All these phases are influenced by global configuration files and individual command line settings. The global configuration files contain the information that the user has to give about the code to be analyzed:

- The cache configurations. On the PowerPC 755, the caches can be partially locked and the amount of locking can be specified by giving the relevant cache size and associativity. Also, this feature can be used to explore the effects that different cache architectures (size, replacement strategy, associativity) would have on the WCET; Section 8.1 contains as an example the comparisons of a real LRU cache against the MPC 755 PLRU cache.

[1]The WCET contribution for basic blocks are not the same as the results of the pipeline analysis for these blocks. Blocks may never be traversed in a worst-case path or they may be traversed more than once.

- The memory configuration. For each memory area in the system, its timings and attributes must be given. Timings give the access times for a memory access in bus cycles. Attributes determine the behavior of an access in this region:
  - If an access to this area is cacheable or not. Some memory areas in a system are not be cached by the processor, e. g. memory mapped control registers, for others one may choose not to cache them for reasons originating from the software design itself (DMA memory areas, e. g.).
  - If this memory area is *guarded*, i. e. data prefetching is not allowed for this area (control registers).
  - Timings and attributes can be given separately for code and data accesses and read or write accesses to make a fine grained attribute control possible.
- The precision that the data-flow analysis should have for all the analyses performed in the toolchain. This is achieved by giving the *context mapping* which should be applied to introduce additional contexts into the data-flow analyses, cf. Section 3.3.1. As the loop iteration bounds are known, the analyses can be configured in such a way that every loop iteration is analyzed separately.
- Constraints for the instructions and control-flow can be annotated in the configuration files. This allows to exclude certain code segments from the analysis or to analyze at a finer granularity than a complete function.
- To make the analysis of isolated functions more precise, intervals for the values of arguments to the function can be specified. This is very helpful to avoid the assumption that a memory access indexed by a function parameter may span the whole memory in the system (including slow peripheral device's control registers). By giving an interval for the possible values of the parameter, the memory accesses can be restricted to the RAM areas, increasing the precision considerably.
- Finally, loop bounds for loops that cannot be detected by the loop analysis can be annotated. These can not only be given with the address of the loop but also annotations like "the second loop in the function runs for at least 3 and at most 10 iterations" can be given[2].

[2]The tool can also support the extraction of certain annotations from comments placed into the source code of the program.

The different phases of the toolchain can further be influenced by command line options that effect only the selected phase.

- The pipeline analysis can be instructed to consider only local worst-cases in its abstract model simulation, reducing the nondeterminism of the model and decreasing the analysis costs. Naturally, the results may not be a WCET bound due to timing anomalies possible with the processor, cf. Section 8.1.

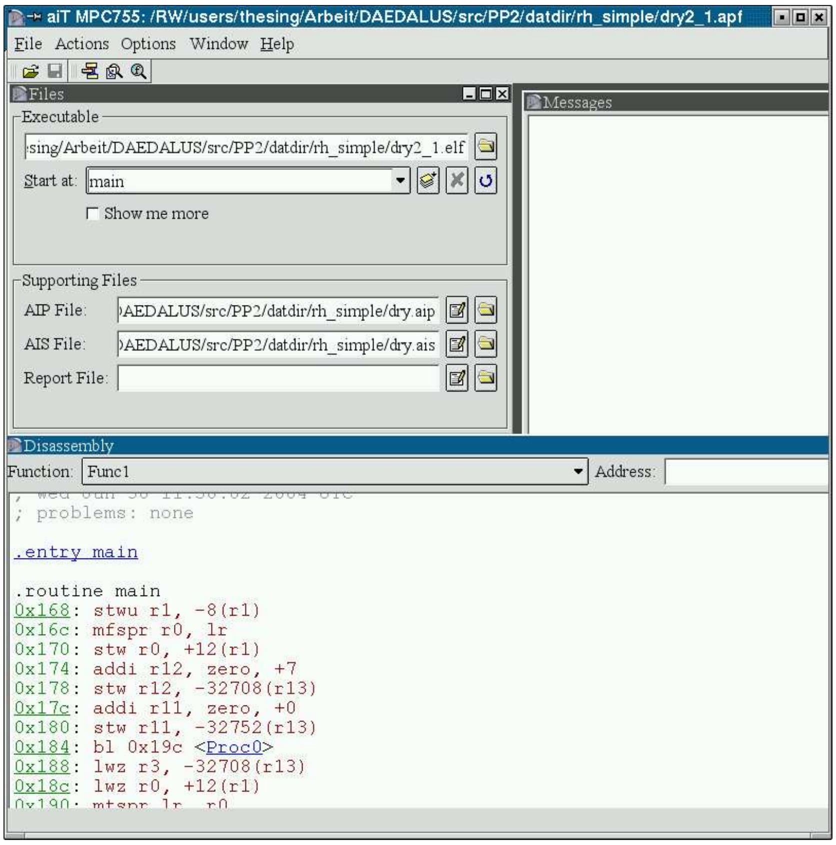
- A special cache mode lets the pipeline analysis assume that every access to a cacheable memory area is a cache hit (for instruction and/or data cache). This allows to investigate an upper bound for the influence of the cache misses.

- In another cache mode, the caches are treated as containing no useful information at all. I.e. for every cache access, the analysis can neither exactly assume a cache miss nor a cache hit and follows both possibilities. This can be used to obtain some information about the benefit of the caching used by comparing the results against a “normal” analysis run.

The main data that the phases of the toolchain share is a textual representation of the CFG of the program to be analyzed, which is generated by the first phase, control-flow reconstruction. This intermediate representation contains the instructions of the program together with attributes for each instruction, grouped into basic blocks. CRL is described in more detail in [Lan98]. Information gained by many of the analyses is merged into CRL in the form of new attributes or transformations on the CRL representation are done, e. g. for loop transformation. This gives a clear interface between different phases of **aiT** and also to additional tools, e. g. the trace verification of Section 7.3. Furthermore the representation is generic so that some programs implementing phases in **aiT** can be shared among different versions of **aiT** or the toolchains of other programs without modifications (e. g. path analysis).

The toolchain of **aiT** is held together by a central driver program that calls the different phases and also provides a graphical user interface within which the configuration can be changed easily. Figure 6.2 shows the main window of **aiT**.

The executable (e. g. in ELF format) and the location of the configuration files can be selected in the upper left window together with the start point in the executable that is to be analyzed. The window on the lower left shows the disassembly of the selected function, while any message produced during the analysis is displayed in the window on the right. The tool can save all settings in a project file that can also be used to perform analyses in batch mode without the graphical user interface to facilitate analysis automation.



aiT MPC755: /RW/users/thesing/Arbeit/DAEDALUS/src/PP2/datdir/rh_simple/dry2_1.apf

Executable: sing/Arbeit/DAEDALUS/src/PP2/datdir/rh_simple/dry2_1.elf
Start at: main
Show me more

Supporting Files
AIP File:)AEDALUS/src/PP2/datdir/rh_simple/dry.aip
AIS File:)AEDALUS/src/PP2/datdir/rh_simple/dry.ais
Report File:

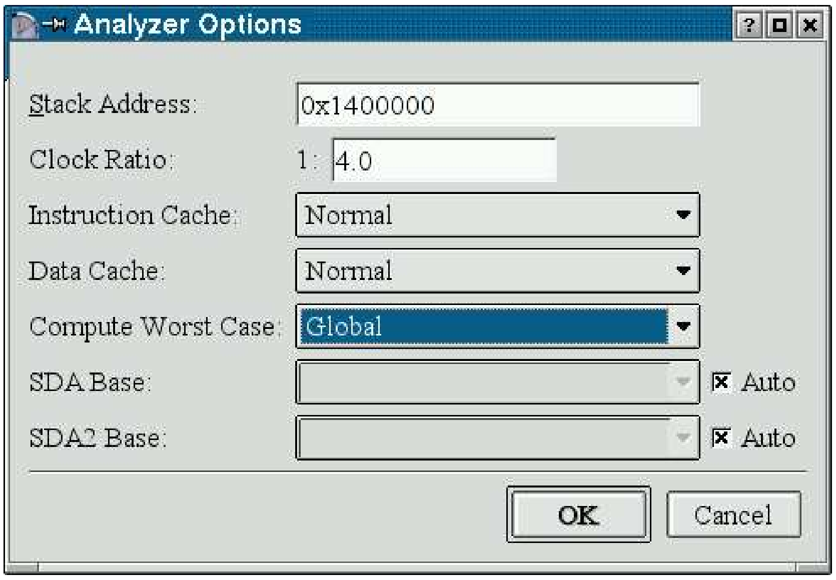
Function: Func1 Address:

```
; problems: none

.entry main

.routine main
0x168: stwu r1, -8(r1)
0x16c: mfspr r0, lr
0x170: stw r0, +12(r1)
0x174: addi r12, zero, +7
0x178: stw r12, -32708(r13)
0x17c: addi r11, zero, +0
0x180: stw r11, -32752(r13)
0x184: bl 0x19c <Proc0>
0x188: lwz r3, -32708(r13)
0x18c: lwz r0, +12(r1)
0x190: mtspr lr, r0
```

Figure 6.2: **aiT**'s main window

Figure 6.3 shows one dialog in which options influencing the different phases can be set directly. Here, the initial value of the stack pointer and the cache modes (cf. above) can be set as well as the ratio of the speeds of the processor core and bus clocks. The PowerPC EABI[3] defines that global data is addressed relative to two registers, which are loaded with fixed addresses during the program prologue. The value of these registers can be inferred directly from the executable's symbol table or given manually. Another switch in this dialog instructs the pipeline analysis to only consider local worst-cases in its analysis. With this, e. g., a cache access not known to be a cache hit is assumed to be a cache miss. The global view would

[3]The **E**mbedded **A**pplication **B**inary **I**nterface fixes the environment in terms of linkage and register usage.



Stack Address: 0x1400000
Clock Ratio: 1: 4.0
Instruction Cache: Normal
Data Cache: Normal
Compute Worst Case: Global
SDA Base: [✓ Auto]
SDA2 Base: [✓ Auto]

Figure 6.3: One of **aiT**'s option windows

follow both alternatives in the analysis.

aiT has three top-level functionalities:

- The reconstructed CFG can be displayed in an interactively explorable representation.
- The WCET analysis can be performed and the results displayed graphically.
- The WCET analysis can be performed and the cycle-wise state evolution of the abstract pipeline model can be displayed in an interactively explorable view.

Examples of the outputs from these three modes are given in Section 6.4.

6.1 Analyzing Binaries

aiT takes as input a binary executable (in ELF or COFF format). While it would be much easier to start the analysis on something at a "higher" level, e. g. assembler or even C source code, this is not possible in our setting: it is the common understanding in the avionics verification area that for timing validation an analysis giving answers about the behavior of an executable (notably its running time) has to be performed on the executable itself. Using source or intermediate code is not feasible as this code undergoes transformations before it ends up in the executable (compiler, code generators,...). Another complication is that it is difficult to use information from the source code (e. g. loop iteration bound annotation made as comments in the C source code) in the analysis itself: if the toolchain that produces code for the avionics system is changed, the *whole* toolchain must be recertified with the airworthiness authorities. Therefore, it is practically impossible to introduce changes into this toolchain in order to facilitate validation. As a consequence, **aiT** only relies on the binary executable and separate annotations made by the user of the tool[4]. Practice has shown that this approach scales up to real-life analysis tasks.

Working on executables also has the advantage that many important parameters are fixed and can be utilized directly in the analyses. E.g. the addresses of the code are known, which would not be the case for assembler or C code. Thus, cache analysis is much easier. Otherwise, modular cache analysis would be necessary which reduces the precision and increases the costs of the framework, cf. [RPTW04].

The details of the reconstruction of control-flow from binaries are described in detail in [The02]. We only highlight some important issues. The reconstructed control-flow must be *safe*, i. e. every execution path possibly taken during a real execution must be present in the CFG. This becomes difficult for machine instructions whose successor(s) are dynamically computed. This feature is often used to implement function calls through a function pointer or, less obviously, to implement high level language constructs equivalent to the **switch** statement in C. Function tables for **switch** statements and similar constructs can often be recognized automatically if the compiler that generated the code is known. This is because a compiler lays out the necessary function tables and call code in a fixed way. The compiler can often be deduced from other information in the binary[5].

If the target of a computed branch or call cannot be determined statically, the CFG reconstruction terminates with an error message. In this case, user annotation in a configuration file is required to specify all possible targets of that instruction. The other alternative is to assume that all routines in the executable may be called by this instruction, which would lead to very imprecise results.

The processor architecture itself can make it difficult to recognize function calls and return instructions. E.g. on the PowerPC architecture, function call and

[4]Other versions of **aiT** are able to extract the annotations made in the source code itself.

[5]The **gcc** compiler defines a special variable in a data section to indicate that the code was generated by it, etc.

return are performed by two machine instructions, **blr** and **blrl**. These instructions continue execution at the address stored in the *link register*. The second form in addition stores the value of the instruction after it in the link register. Here, the CFG reconstruction must make sure that the link register has been loaded with the address of a function and identify the function, e. g. by using pattern matching on the code. To complicate things, the link register may also be used for indirect or far function calls and to implement **switch** statements.

The result of the CFG reconstruction is the CFG in CRL representation, which is written to a file. Attributes for each instruction in the CFG are added by this phase. These attributes define entities like the registers used by an instruction, its opcode, class, etc. Later phases make heavy use of the information computed in this phase. In addition, the call graph is present in the CRL file as special attributes for basic blocks: which block may call another block.

6.2 Value Analysis

The value analysis phase computes intervals as safe approximation to the addresses that may be used during data accesses[6]. In [Sic97] M. Sicks described a value analysis for SPARC machine code that computes safe approximations for the contents of machine registers. The principle is to perform the interval analysis from [CH78] on machine programs. The value analysis performs an abstract interpretation of the machine instruction semantics over the domain of intervals. The exported result is not an interval for the contents of each machine register for each program point, but only those that access data memory. Here, the address calculation of the instruction together with the internally present interval information is used to compute a safe interval for the address that is used during the memory access[7]. This analysis uses the same context separation technique as the pipeline analysis, in order to analyze loops or nested call more precisely. As a consequence, memory access information is produced for each context of an instruction and written to a result file.

As the value analysis has to compute information about the outcome of instructions influencing conditional branches[8], it is often possible to determine that a conditional branch will never branch to one of its two successors (fall-through successor or taken-branch successor). In this case we know that the code starting at that untaken successor is *infeasible*, i. e. will never be executed. This happens very often if the analysis distinguishes enough context to separately analyze loop iterations. As this information is also very important for the pipeline analysis, it is

[6]The instruction accesses are known and fixed by the structure of the CFG.

[7]Or accesses, as, e. g. the ColdFire can perform a read and write with one instruction.

[8]With other words, about the *status register* of the processor.

written to the result file in addition to the the intervals for data memory accesses. In the pipeline analysis this infeasibility information is used in order to decide if a predicted branch was resolved correctly or that a pipeline state can safely be deleted during analysis if it represent execution of infeasible code.

In [FHL^{+}01] results for the precision of the value analysis results are presented. These results are depicted in Table 6.1.

Task	reads [%]			writes [%]			infeasible	time
	exact	good	?	exact	good	?	[%]	[s]
1	93.32	3.69	2.99	94.51	4.11	1.38	9.1	61
2	93.42	4.01	2.57	93.39	4.62	1.99	8.5	24
3	92.58	4.20	3.22	93.25	4.59	2.16	9.3	31
4	90.24	5.61	4.15	91.71	6.35	1.94	13.5	21
5	93.79	3.33	2.88	94.61	3.83	1.56	7.0	59
6	93.23	4.21	2.56	93.02	4.82	2.16	10.2	26
7	92.06	4.63	3.31	93.13	5.01	1.86	11.0	35
8	90.87	5.03	4.10	91.97	5.88	2.15	11.4	18
9	93.97	3.22	2.81	94.74	3.72	1.54	6.8	56
10	92.40	4.72	2.88	92.79	5.32	1.89	11.0	26
11	92.16	4.47	3.37	92.98	4.90	2.12	9.4	31
12	90.51	5.28	4.21	91.57	6.16	2.27	11.5	24

Table 6.1: Value analysis results

These results are for the ColdFire 5307 and were obtained on a 1GHz Athlon. The classification are for all contexts with the following meanings:

- **exact** means that the address is exactly known, i. e. an interval with exactly one member
- **good** means that the interval spans at most 16 cachelines or 256 bytes
- **?** means that that the interval is larger than 256 bytes
- **infeasible** gives the percentage of instructions who were determined to never be executed as stated above
- **time** gives the time for the analysis

The results for the PowerPC are of comparable quality. Naturally, the precision heavily depends on the code structure of the analyzed executable. The avionics code is very friendly for the analysis, the results will be less precise for hand written or highly optimized assembly.

6.3 Separating Analyses

At this point it should be stressed that the different phases of the toolchain implement different kinds of analyses. While these analyses could in principle be integrated into one big analysis, it is always desirable to separate them for performance and engineering reasons. While one may loose some precision by this separation, the gain in efficiency of the whole tool and the reduced design complexity is worth the effort. How much precision is lost by the separation depends on the interactions of the components that are the main focus of the individual analyses. While nearly no precision is lost by separating value analysis from the pipeline analysis, this may not be the case for other analyses. In former publications (e. g. [FKL$^+$99]) we considered a separate cache and pipeline analysis to reduce the complexity of the pipeline analysis. However, at least for the processors we implemented **aiT** for, the cache is not independent from the pipeline. Both the ColdFire 5307 and the PowerPC 755 implement branch prediction, which redirects fetching by some heuristics to reduce stall time due to control hazards. For both processors, the fetch machinery and execution logic are independent. The latter resolves the branch condition upon which is being speculated while the former fills the prefetch queue with instructions, altering cache contents and/or ages of blocks in the cache. The amount of prefetching done depends on the speed of the execution pipeline, thus its state and so the results of the pipeline analysis. The pipeline analysis, on the other hand, depends on the results of the cache analysis, i. e. cache access classifications and times. To break this interdependency, one has to make conservative approximations in the cache analysis to account for the prefetch that may (but need not) happen due to branch prediction.

Figure 6.4, taken from [HLTW03], gives some results for the loss of precision incurred due to a separate cache analysis. Here, the conservative approximations are depicted as **APPROX**, while the results taken from the integrated cache and pipeline analysis are shown as **PURE**. The measure shown for twelve example programs is the number of cachelines known to be in the cache divided by the number of program points and contexts, cf. Section 8.1. The ratio between the measures lies between 1.3 and 1.484. The benchmarks each had around 40kB of code. The loss of precision is significant, especially if one considers that the effects of data accesses on the unified cache have not been accounted for. Thus, we decided in this case to do a more complex integrated cache and pipeline analysis.

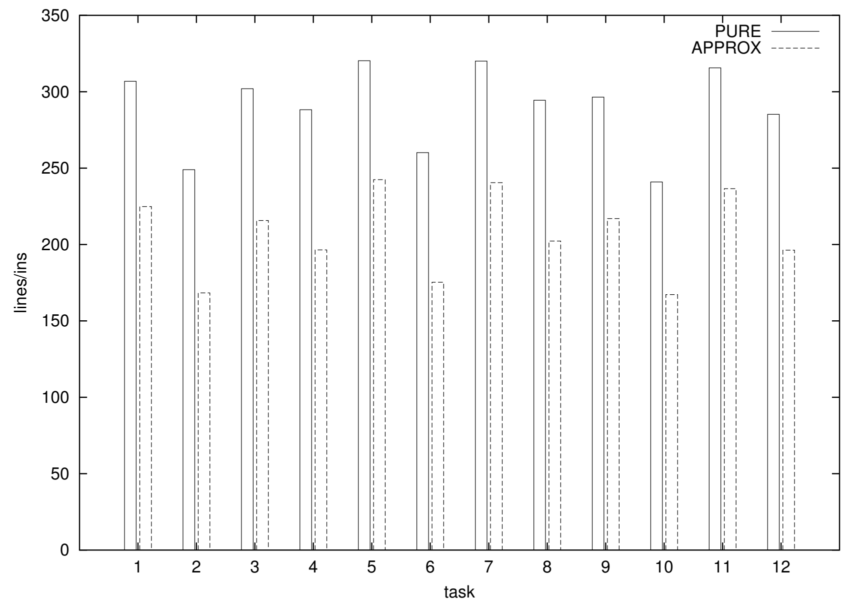


Figure 6.4: Precision loss due to separate cache analysis

The same arguments and conclusions apply to the cache of the PowerPC and the respective analyses.

Separating analyses is made easy in the framework developed by AbsInt, as data can be merged into the intermediate CRL description or written out to separate result files and easily be read back into subsequent phases and associated with instructions and contexts.

6.4 Visualization

aiT includes an interactive visualization of the program and WCET analysis results that is based on the powerful features of the **aisee** graph visualization tool [aiS].

One functionality of **aiT** is the visualization of the reconstructed CFG, as depicted in Figure 6.5. The CFG is first shown as the global call graph, connecting functions together (lower half). Each function box can be unfolded showing the basic blocks of the program and the instructions together with the control-flow edges connecting them. As can be seen, loops are extracted into procedures to

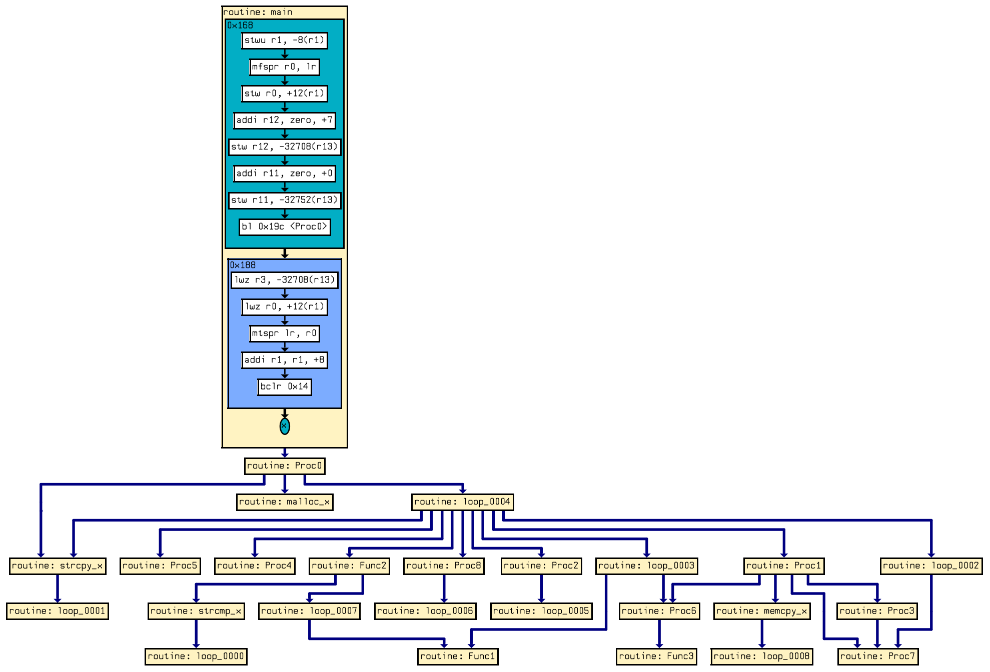


Figure 6.5: CFG reconstructed by **aiT**

facilitate a more detailed analysis.

After computation of the WCET, the results are also shown graphically as in Figure 6.6.

Apart from the overall WCET in the red box at the top, the CFG is shown again with the depiction of one worst-case path, i. e. one path along which the WCET will be reached. This path is shown in red color. Again, the routine boxes can be unfolded showing the basic blocks and the information obtained for them. At the edges connecting basic blocks, the contribution of execution along that edge for the block at which the edge originates and the number of traversals of that edge along the worst-case path are shown. The contribution of each function to the WCET can also be displayed (not shown in the figure). This information gives a detailed view about the WCET and its distribution among the functions and basic blocks of the program. This information can be very useful if the computed WCET differs from the expected one or is greater than the deadline of a task.

The third mode of the tool allows to follow the cycle-wise simulation of the abstract pipeline model. It contains all information that is computed by the analysis. This view is selected for a given basic block and context (e. g. the third iteration

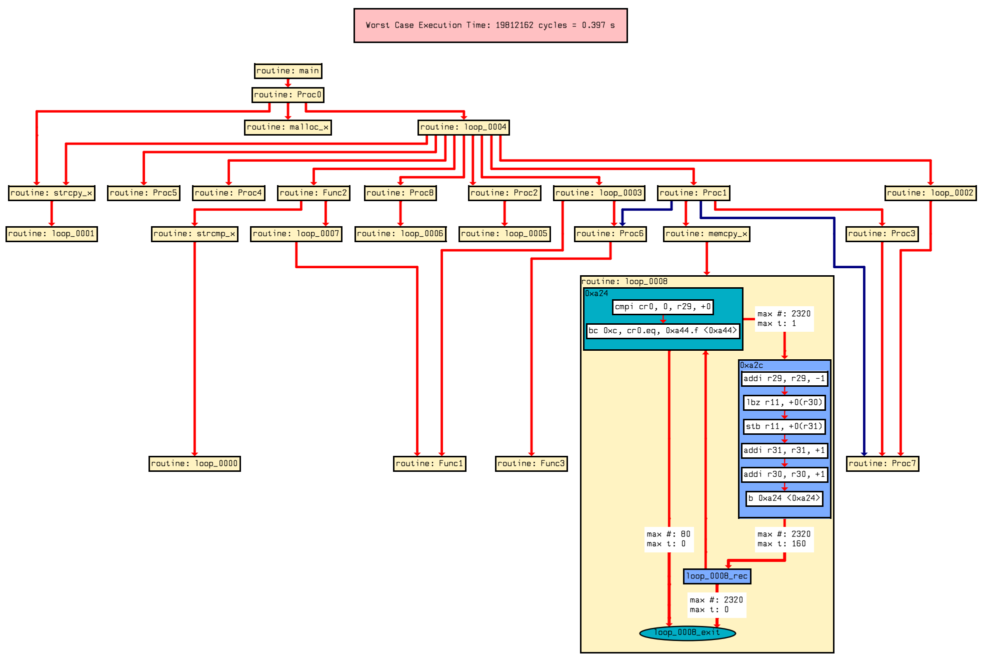


Figure 6.6: Result visualization

of a loop, etc). An overview of the information displayed is shown in Figure 6.7.

The whole figure corresponds to one instruction in the selected basic block. The darker (red) boxes at the top are the starting states for the simulation, the darker (blue) ones near the bottom are those states in which the instruction has left the pipeline and thus finished its execution. These states are then the starting states for the next instruction in the basic block (or the first instruction in a another basic block). Each row in the figure stands for one cycle of simulation. The nondeterminism of the abstract model is already evident in the figure: in the second row, the starting states split into up to four successor states. In this case, this is because it is not known if the instruction fetch of an instruction is a cache hit or miss. And as the fetch happens to cross a cache line boundary[9] there are four possible successor states: two cache lines involved and for each line cache hit and cache miss; the analysis follows all possibilities. This state expansion is reduced by the fact that often pipeline states coming from different columns in the figure collapse into the same end state because they are identical. This effect can very often be seen at the end of accesses to external memory as these take so

[9]The PowerPC 755 fetches up to four instructions in one block.

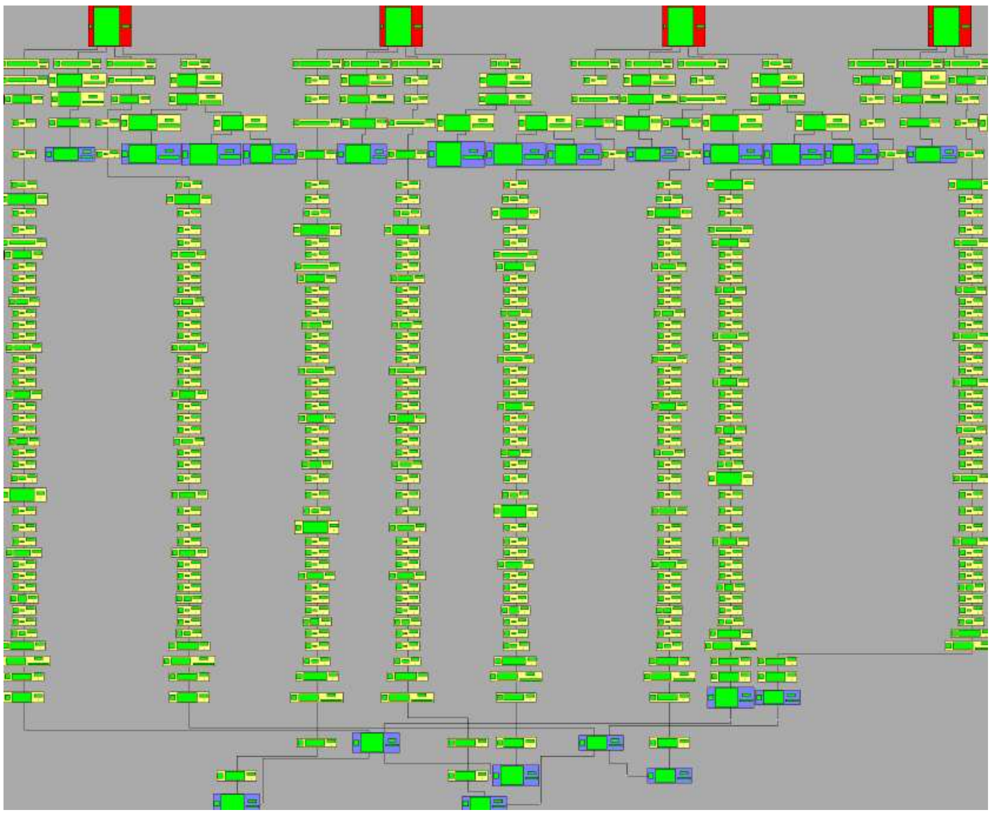


Figure 6.7: State exploration

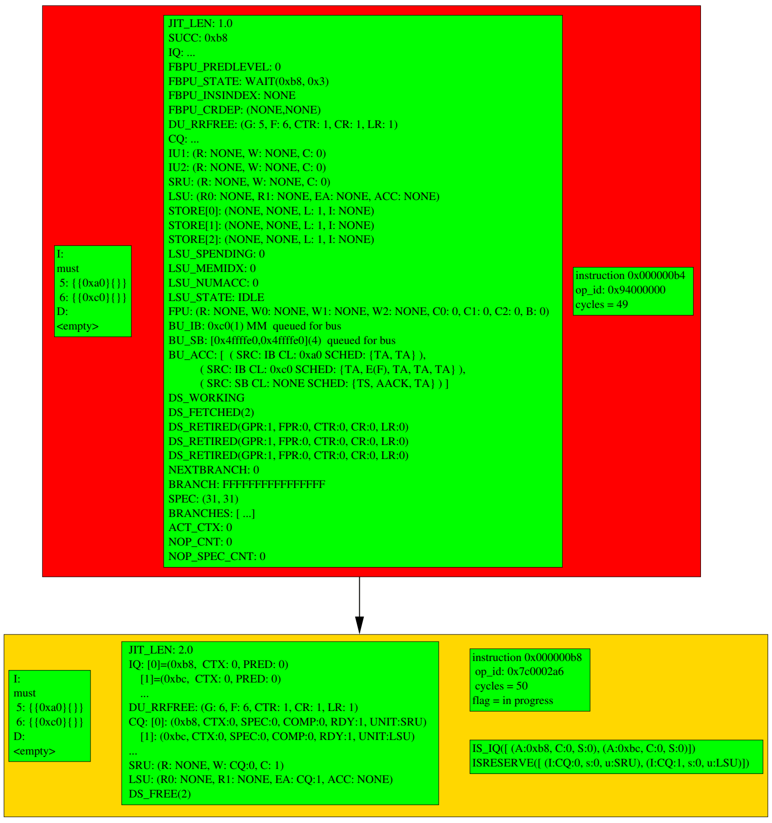
long that all instructions in the pipeline have finished execution and the pipeline is empty.

A more detailed view for one of the pipeline states from Figure 6.7 is shown in Figure 6.8.

The information in the boxes is divided into three parts:

- the cache analysis information is displayed in the box on the left.
- The box on the right contains information about the instruction currently being executed[10], e. g. the opcode and some instantaneous signals, if asserted.
- In the center, the pipeline state is shown, including in this figure the values of all components of the abstract pipeline model, as defined in Section 4.4.

[10]I.e. the node in the CFG were we are iterating the analysis simulation.



```
I:
must
 5: {{0xa0}{}}
 6: {{0xc0}{}}
D:
<empty>
```

```
JIT_LEN: 1.0
SUCC: 0xb8
IQ: ...
FBPU_PREDLEVEL: 0
FBPU_STATE: WAIT(0xb8, 0x3)
FBPU_INSINDEX: NONE
FBPU_CRDEP: (NONE,NONE)
DU_RRFREE: (G: 5, F: 6, CTR: 1, CR: 1, LR: 1)
CQ: ...
IU1: (R: NONE, W: NONE, C: 0)
IU2: (R: NONE, W: NONE, C: 0)
SRU: (R: NONE, W: NONE, C: 0)
LSU: (R0: NONE, R1: NONE, EA: NONE, ACC: NONE)
STORE[0]: (NONE, NONE, L: 1, I: NONE)
STORE[1]: (NONE, NONE, L: 1, I: NONE)
STORE[2]: (NONE, NONE, L: 1, I: NONE)
LSU_SPENDING: 0
LSU_MEMIDX: 0
LSU_NUMACC: 0
LSU_STATE: IDLE
FPU: (R: NONE, W0: NONE, W1: NONE, W2: NONE, C0: 0, C1: 0, C2: 0, B: 0)
BU_IB: 0xc0(1) MM  queued for bus
BU_SB: [0x4ffffe0,0x4ffffe0](4)  queued for bus
BU_ACC: [  ( SRC: IB CL: 0xa0 SCHED: {TA, TA} ),
         ( SRC: IB CL: 0xc0 SCHED: {TA, E(F), TA, TA, TA} ),
         ( SRC: SB CL: NONE SCHED: {TS, AACK, TA} ) ]
DS_WORKING
DS_FETCHED(2)
DS_RETIRED(GPR:1, FPR:0, CTR:0, CR:0, LR:0)
DS_RETIRED(GPR:1, FPR:0, CTR:0, CR:0, LR:0)
DS_RETIRED(GPR:1, FPR:0, CTR:0, CR:0, LR:0)
NEXTBRANCH: 0
BRANCH: FFFFFFFFFFFFFFFF
SPEC: (31, 31)
BRANCHES: [ ...]
ACT_CTX: 0
NOP_CNT: 0
NOP_SPEC_CNT: 0
```

```
instruction 0x000000b4
op_id: 0x94000000
cycles = 49
```

```
I:
must
 5: {{0xa0}{}}
 6: {{0xc0}{}}
D:
<empty>
```

```
JIT_LEN: 2.0
IQ: [0]=(0xb8,  CTX: 0, PRED: 0)
    [1]=(0xbc,  CTX: 0, PRED: 0)
    ...
DU_RRFREE: (G: 6, F: 6, CTR: 1, CR: 1, LR: 1)
CQ: [0]: (0xb8, CTX:0, SPEC:0, COMP:0, RDY:1, UNIT:SRU)
    [1]: (0xbc, CTX:0, SPEC:0, COMP:0, RDY:1, UNIT:LSU)
...
SRU: (R: NONE, W: CQ:0, C: 1)
LSU: (R0: NONE, R1: NONE, EA: CQ:1, ACC: NONE)
DS_FREE(2)
```

```
instruction 0x000000b8
 op_id: 0x7c0002a6
 cycles = 50
flag = in progress
```

```
IS_IQ([ (A:0xb8, C:0, S:0), (A:0xbc, C:0, S:0)])
ISRESERVE([ (I:CQ:0, s:0, u:SRU), (I:CQ:1, s:0, u:LSU)])
```

Figure 6.8: State exploration detail

For subsequent cycles, only the information that changed is displayed in order not to clutter the view with irrelevant details. Naturally, the whole pipeline state can be retrieved for these states too by unfolding the box in the middle.

All visualizations can be explored by moving around in the view, zooming and folding or unfolding boxes. The information can also be exported to other formats for further processing.

In addition to the visualizations, **aiT** also produces a textual form of the results that is easier to integrate into batch jobs for the automated analysis of a sequence of programs.

6.5 Practical Results

The **aiT** versions for the ColdFire 5307 and the PowerPC 755 have been designed and refined for the needs of Airbus France. As has been stated earlier, the details of the hardware that the programs to analyze will run on have a tremendous influence on the results of the WCET tool. Thus, comparing results against actual run-time measurements on real hardware is not meaningful if the hardware that the measurements are performed on differs from the hardware that the analyses model. The target application for the tool is flight critical avionics software (primary flight control), thus the hardware that the analyses model is not available outside of Airbus France. In the course of the DAEDALUS project, a detailed assessment of the WCET results delivered by **aiT** was performed by Airbus France, cf. Section 9.2. The assessment was done using large benchmarks typical for the intended target domain. Due to the sensitive nature of the software under analysis most of the results are not available for publication. Nonetheless, in [TSH^{+}03], a small subset of the evaluation was published. Table 6.2 shows a comparison for the ColdFire version of the WCET tool.

Task	Airbus' method	**aiT**'s results	precision improvement
1	6.11 ms	5.50 ms	10.0 %
2	6.29 ms	5.53 ms	12.0 %
3	6.07 ms	5.48 ms	9.7 %
4	5.98 ms	5.61 ms	6.2 %
5	6.05 ms	5.54 ms	8.4 %
6	6.29 ms	5.49 ms	12.7 %
7	6.10 ms	5.35 ms	12.3 %
8	5.99 ms	5.49 ms	8.3 %
9	6.09 ms	5.45 ms	10.5 %
10	6.12 ms	5.39 ms	11.9 %
11	6.00 ms	5.19 ms	13.5 %
12	5.97 ms	5.40 ms	9.5 %

Table 6.2: Comparison between Airbus' legacy method and **aiT**

Twelve benchmarks have been analyzed by Airbus' personell and compared to the results achieved by a method in practical use at Airbus to compute WCETs for a long time.

The results obtained by **aiT** are more precise than the results of Airbus' legacy method by up to 13.5% and never worse. The analyses were performed on a 1GHz Athlon system with 1GB of main memory. Running times of the analyses were below 30 minutes. The analysis results were also compared against real execution times, finding that the predicted WCET is always an upper bound for these times.

We did not perform extensive measurements on "similar" hardware or integrate a modeling of the hardware we have available because of the following reasons:

- When measuring on hardware that differs from the hardware described by the model, one has to add a margin to the results that accounts for the effects of the differing hardware. In a few experiments, we found that this margin is quite large compared to the WCET results so that an error of the analysis can barely be found; furthermore, no meaningful statement about the precision of the analysis can be made because of this margin.

- Making models for two rather complicated processors and peripherals is a time consuming task. Implementing the model and tuning and verifying the results is a costly process. Therefore, we did not want to invest into a hardware model[11] that resembles the hardware we had available in order to run larger benchmarks. Besides, no large publically available benchmarks exist that have a similar size as the benchmarks used by Airbus France for their evaluation of **aiT**.

It is often asked to compare the results of a WCET analysis against those of other tools. We think that this is impossible due to the following reasons.

- There is no such thing as a realistic standard benchmark for our application area. There are a few benchmarks with collections of small functions, like matrix multiplication or fast Fourier transformation. Computing times for these benchmarks is easily possible but has no significance for answering the question wether the method scales up to real-life programs. Also, from our experience real-life benchmarks contain a lot more than these simple functions.

- The processors used in different approaches are not comparable. E.g. computing WCET for a deterministic DSP without caches and no timing variant components is easy and can be done quite precise. Computing the same information for the PowerPC 755 is much more complex and the precision cannot be expected to be as precise.

[11]More precisely, a model of the peripherals, in this case.

- Other work uses smaller benchmarks and already encounters complexity problems. We chose to let the benchmarking of our tool be performed by a third party with the benchmarks they classify as representative for the complexity and size of real-life programs.

- One goal of the ARTIST2 Network of Excellence of the European Union is the development of a realistic set of benchmarks for real-time applications.

Section 9.2 contains some conclusions about **aiT** from the reviewers of the DAEDALUS project who had wider access to the results of the Airbus assessment.

Chapter 7

Verification of the Analysis

Sed quis custodiet ipsos custodes?
Decimus Junius Juvenalis, Satura VI

The WCET analysis is used to obtain the bounds of each tasks execution time needed for the schedulability analysis. The scheduling analysis is performed to *validate* that the system runs within the timing bounds specified in its design. As such, even a wrong scheduling analysis and also a wrong WCET analysis do not *introduce* errors into the system, they just fail to *detect* errors already in the system. This does not mean that WCET analysis is useless, rather it limits the effort needed to prove the absolute absence of bugs in the analysis in practice. Other tools, e. g. code generators, have to be validated more thoroughly, because they can indeed introduce bugs into the system by generating wrong code[1].

As we have said earlier, our methods for pipeline analysis (and the other components of the **aiT** framework, cf. Chapter 6) are based on provably correct methodologies. Thus, verifying the correctness of **aiT**'s results seems to be easy. In fact, there are a number of sources for possible bugs in the tool:

- Implementation errors can never be ruled out completely, as the toolchain components are (partially) written by-hand, especially the implementation of the abstract pipeline analysis is done in hand-coded C.

- The pipeline analysis is only correct w.r.t. the concrete model that is the starting point of the abstractions and analyses. If this model does not describe reality, then also the analyses will not analyze reality and the results

[1]DO-178B [DO192], the standard governing the software development process for avionics contains only short passages about analysis tools, while much more is said about code generating tools.

will probably be invalid. The next section deals with this problem in more detail.

- **aiT** uses an ILP to find the worst-case path and time of the whole program. ILP solvers are known to sometimes produce incorrect results due to the fact that the ILP is relaxed to the domain of floating point numbers and a solution is searched for in this domain. Numerical instabilities in the computations may lead to a wrong solution being found. In the AVACS project, methods will be developed to verify the correctness of solutions returned by ILP solvers. Another idea is to make use of the special form of ILPs generated for the WCET problem and use purely integral techniques to solve them, avoiding the numerical instabilities of currently available ILP solvers.

The next section will describe the sources used for the pipeline models in this thesis and possible problems w.r.t. analysis result validation coming from them. In Section 7.2 the steps undertaken to verify the correctness of aspects of the modeling process are presented. This chapter closes with some ideas about the automatization of the validation process in Section 7.3.

7.1 Modeling Sources

As stated in Chapter 4, we did not have authoritative HDL models available for the two processors for which **aiT** had to be instantiated first. Therefore, the models were made using three sources of information:

- The processor's handbooks,
- experiments performed on the actual hardware,
- communication with the design teams of the processors and the support teams of the manufacturer.

7.1.1 Processor Handbooks

The models were designed starting from the descriptions in the processors' handbooks [PPC97a] and [Mot00]. Handbooks contain a varying degree of information about the processor's internal design. Neither for the ColdFire, nor the PowerPC the presented information was sufficient to obtain the processor model with a satisfying degree of precision. E.g. the description of the cache replacement and design for the ColdFire in [Mot00] does not state explicitly if there is one cache replacement counter for all sets or one counter per cache set. Other information is stated differently at different places in the handbooks, contradicting each other.

However, the basic structure of the processor's implementation and thus the basic structure of our models is described in the handbooks.

7.1.2 Experiments on the Hardware

To decide unclear or unexplained situations in the processor's program execution, we designed test programs and executed them on the actual hardware. These experiments were designed such that the program's result differs for the different possible scenarios. E.g., to solve the question, if the ColdFire has one replacement counter per set or not, we designed the following experiment and program:

1. The cache is invalidated, clearing all lines and marking them as invalid.
2. The cache is filled with data by performing 512 accesses to addresses 0x0, 0x10, ..., 0x1FF0 (in hexadecimal notation), giving the contents of Table 7.1.
3. 128 more accesses are performed to the addresses 0x2000, 0x2010, ..., 0x27F0. If there is one replacement counter for each set, then the cache contents is as in Table 7.2 and the counter is 1 for every set. If there is only one global counter, then the contents is as in Table 7.3. In both tables, the elements newly placed into the cache are denoted in boldface.

	Way			
Set	0	1	2	3
0	0x0000	0x0800	0x1000	0x1800
1	0x0010	0x0810	0x1010	0x1810
...	...			
126	0x07E0	0x0FE0	0x17E0	0x1FE0
127	0x07F0	0x0FF0	0x17F0	0x1FF0

Table 7.1: Cache contents after 512 accesses

After this setup phase, two different access patterns for 128 additional accesses were measured for execution time:

1. Accessing the elements 0x00, 0x10, 0x20, i. e. the elements in way 0 of Table 7.1.
2. Accessing the elements 0x00, 0x810, 0x1020, 0x1830, 0x40, i. e. the elements in the diagonals of Table 7.1.

	Way			
Set	0	1	2	3
0	**0x2000**	0x0800	0x1000	0x1800
1	**0x2010**	0x0810	0x1010	0x1810
...	...			
126	**0x27E0**	0x0FE0	0x17E0	0x1FE0
127	**0x27F0**	0x0FF0	0x17F0	0x1FF0

Table 7.2: Cache contents, 640 accesses, per-set counter

	Way			
Set	0	1	2	3
0	**0x2000**	0x0800	0x1000	0x1800
1	0x0010	**0x2010**	0x1010	0x1810
...	...			
126	0x07E0	0x0FE0	**0x27E0**	0x1FE0
127	0x07F0	0x0FF0	0x17F0	**0x27F0**

Table 7.3: Cache contents, 640 accesses, global counter

If there is only one global replacement counter, then the first access pattern will produce 32 cache misses for the elements 0x00, 0x40,.... It produces 96 cache hits for the elements 0x10, 0x20, 0x30,0x50, 0x60,.... The second access pattern will produce 128 cache misses in this case, as all the diagonal elements have been replaced by the elements 0x2000, 0x2010,.... If there is one counter per set, then the first access pattern will produce 128 cache misses, as all elements in way 0 had been replaced. For the second pattern, one would observe 32 cache misses for the accesses to 0x00, 0x40, ...and 96 cache hits for the accesses to 0x810, 0x1020, ... As a cache miss takes around 8 bus cycles on our MCF 5307 board, or 32 internal cycles, while a cache hit should be served in 2 internal cycles, the time for the complete access loop differs enough to be measurable with the built-in timers for both scenarios (32 misses + 96 hits vs. 128 misses).

To obtain a time bound for the memory access times, the experiment was performed with caching disabled, too.

Performing the experiment for both patterns shows times consistent with the assumption that only one global counter exists.

Performing these experiments is difficult, as other influences, e. g. instruction fetching must be avoided as far as possible. Our solution was to place the code for the experiment into the internal SRAM of the MCF 5307, which is operated independently of the external bus machinery, which was exclusively available for the data accesses.

For the PowerPC 755 some details were validated or investigated by experiments performed along the same lines. E.g. the handbook explicitly selects one instruction to be used as a “no operation” instruction, namely the `ori r0,r0,0` instruction. On the architectural state, this instruction has clearly no effect. For timing, it is important, if this instruction is dispatched to an integer unit and does nothing there, or if the dispatcher already throws away the instruction, never dispatching it to a functional unit. Also, if the instruction is thrown away, it is important to know, if it is thrown away before it reaches the two dispatchable positions 0 and 1 in the instruction queue or it is simply discarded during the normal dispatch from those entries.

An experiment has been designed that executes a special instruction sequence that changes the dispatching of integer instruction to the IU1 and IU2 units, causing some divide instructions to stall if nops are discarded and not to stall, if they are dispatched to a unit. Execution times (of this repeated sequence) were measured with the internal performance counters of the MPC 755, counting processor cycles. The experiments validated the behavior that nops are discarded from the normal dispatch entries and are simply thrown away.

Designing and performing these experiments increases the costs of modeling further. Therefore, we decided to only perform experiments if a behavior was in doubt judging from the documentation or if there had been strong evidence that a behavior was different from the documented one.

7.1.3 Other Sources

Apart from the information in the handbooks and the experiments performed, we forwarded detailed questions about the processor’s behavior to the support hotline of Motorola. These questions were in most cases forwarded to the design team of the PowerPC 755 and answered by them, guaranteeing a sufficient level of authoritative information. Naturally, this process was very time consuming and increased the modeling effort considerably.

7.2 Verifying the Modeling

Not all aspects of the model had been verified by experiments or by information from the design teams of the processors. Thus, there was still a lot of space for errors in the model. To further validate the model, we used three methods:

- *Feature Verification*: to verify that the model reproduces certain features, e. g. the number of branch mispredictions is correctly covered by the model, one can design experiments that use the performance counters of the chips to count the relevant events.

- *Local Verification*: small example programs that can be simulated deterministically even in the abstract model can be measured to verify the cycle-precise equivalence of model and real hardware.

- *Global Verification*: longer programs, which can no longer be deterministically simulated in the abstract model can be analyzed and measured on the real hardware. The predicted execution time by the abstract model should be an upper bound to the observed runtimes.

This process is naturally very time consuming as for the first two methods, experiments have to be designed and performed. Besides the design of the programs to use in the experiments, also undesired influences have to be ruled out. E.g. it must be made sure that the whole program code is already cached to minimize the effects of instruction caching on the timing: an external fetch takes so long that smaller internal effects are hidden by it. Thus, the next section contains some ideas about a method by which the necessary effort for validation can be reduced.

7.3 Automation

Validation of an analysis against real executions of test programs can only facilitate the data that is *observable* with the real hardware. Internal processor state can not be observed without intrusion in most systems. Even the use of internal debug facilities like JTAG may influence the execution of the processor. We will restrict ourselves to the observation of the *events* on the external processor bus. Here, a logic analyzer can be connected that samples the signals on the bus and is able to collect a *trace* of events on the bus during the execution of the program. This trace may contain events like “transfer start”, “address acknowledge”, “data word transfered”, etc with the corresponding attributes (addresses, direction, . . .). It can then be checked if this trace is covered by the predictions made by the pipeline analysis.

The pipeline analysis is nondeterministic in the sense that if a cycle update in the abstract model depends on abstract components, the analysis will follow *all* possibilities and computes all possible successor states. Thus, all possible executions of the program are covered by the analysis, not only the *worst* one.

Also, the abstract model and thus the pipeline analysis can be augmented to record the same events in a state that can be observed by a logic analyzer in a real system. If the analysis finds out that a transfer is put on the bus, a "transfer start" event can be recorded with the corresponding pipeline state, giving a set of events for a set of pipeline states.

Given a trace of real events and a set of events predicted by the pipeline analysis for each instruction of the CFG, the validation effort is now reduced to showing that there is a path through the CFG, along which we can see all events in the trace predicted. If a trace is not covered this way, we have found an error. Repeating this process for a number of traces increases the confidence one can have into the correctness of the analysis. Practically, it is not possible to collect the whole execution of some program with a logic analyzer due to technical reasons. Thus, a prefix of program execution must be considered.

More formally, we define

Definition 7.3.1 (Events, Trace): We assume that a set Ev of events $\phi \in \text{Ev}$ is given. The empty event $\varepsilon \notin \text{Ev}$ is used when we want to denote that no event occurs; we define $\text{Ev}_\varepsilon := \text{Ev} \cup \{\varepsilon\}$. A trace $t \in \text{Ev}^+$ is non-empty sequence of events. The pipeline analysis delivers a mapping $\xi : E \to \mathcal{P}(\text{Ev})$ of edges in the CFG to set of events happening when execution goes from the source to the target node of that edge. This mapping is called predicted events. ■

Given a trace $t = \phi_1.\phi_2.\ldots.\phi_l$ of length l, which represents the events observed during (a prefix of the) execution of program G, we want to find a path through the CFG, starting at s_G up to some node n that has an assignment of all events in t to the predicted events along the path. For this, we define the notion of a *match*:

Definition 7.3.2 (Match): Given a trace t and and predicted events ξ, a match of t with ξ is a sequence $m \in (E \times \text{Ev}_\varepsilon)^+$ of edge, (possibly empty) event pairs. A match $m = (e_1,\phi_1).(e_2,\phi_2).\ldots.(e_k,\phi_k)$ is *feasible* if it satisfies $R(\phi_1.\ldots.\phi_k) = t$ and $e_1.\ldots.e_k$ is a path through the CFG of the program G that starts at s_G. Here, $R : \text{Ev}_\varepsilon^* \to \text{Ev}^*$ is defined by

$$R(\phi_1.\ldots.\phi_n) = \begin{cases} \varepsilon & \text{, if } n < 1 \\ R(\phi_2.\ldots.\phi_n) & \text{, if } n \geq 1 \land \varepsilon = \phi_1 \\ \phi_1.R(\phi_2.\ldots.\phi_n) & \text{, otherwise} \end{cases}$$

■

A naive algorithm to find a match simply starts at s_G and tries to match the first event from the trace t against the set of events in the predicted events for the outgoing edges of the node. If the event is found, then the trace t is shortened by its first element and matching continues at the node corresponding to the edge e for which the event was found in $\xi(e)$. The edge e and event are recorded. If the event was not found, an empty event ε is assumed and the search has to continue in all successor nodes, recording all outgoing edges and ε as possible match. In case that the match cannot continue, because all events reachable from that node even over empty events are not compatible with the event at the head of the trace, backtracking has to be performed.

Currently, a master thesis is under way to implement this automatic validation of hardware traces, taking the textual output of a logic analyzer, searching for the desired events in it and building the event trace. Then, this event trace is matched against the results of the pipeline analysis, which writes out all predicted events during program analysis.

In extension to the above definitions, also the distance in time of the events can be considered in trace matching. This may be complicated by the fact that a logic analyzer is not always capable of recording timing exactly enough.

With all the approaches sketched in this chapter, we hope to improve the confidence in the correctness of the analysis results delivered by **aiT**. It may be noted that the question of verifying WCET analysis results against reality under stringent conditions (like DO-178B) has not been considered in previous work to the best of our knowledge. Nonetheless, it forms an important part in the process of introducing static analysis methods into the validation procedures for avionics software.

Chapter 8

Predictability of Modern Hardware

Das Übel erkennen heisst schon ihm teilweise abhelfen

Otto von Bismarck

The pipeline analysis introduced in the previous chapters is based on quite complex hardware models. The complexity is needed to precisely model all the interactions of the different components in the system correctly. This chapter gives an overview over the sources of the interactions and the consequences they have for the complexity and precision of the analysis.

Some hardware features lend themselves not easily to static analysis of their behavior. On the other hand, the analysis itself may loose precision and cause higher costs due to the abstractions it makes to obtain the abstract pipeline model. The next section elaborates these issues and in Section 8.2 some thoughts about the *predictable* design of real-time systems are summarized.

8.1 Sources of Complexity and Imprecision

There are subtle interdependencies between the complexity of an analysis, its cost and the precision that can be obtained by it. A less complex analysis that does not model, e. g., the cache contents precisely must make conservative assumptions in the case of memory accesses. If local worst-cases can be utilized, many cache misses will be assumed by the analysis, increasing the predicted WCET and leading to a loss of precision. Unfortunately, local worst-case decisions can rarely be made, as demonstrated for the two processors handled in this thesis. Here, all possibilities must be considered and the less complex analysis has to consider both cache hit and cache miss scenarios, increasing its costs while *probably* leading to a less precise bound for the WCET[1]. Furthermore, a less complex analysis will probably not be able to ensure that the local worst-case is also the global worst-case in some scenarios. The **aiT** tool for the PPC 755 has been augmented with a run-time option to only consider local worst-cases in its computations. Table 8.1 gives some comparisons for the running time of the WCET analysis with and without this option and the differences of the predicted WCET bounds. It can be seen that indeed the local worst-case leads to a *lower* bound for the WCET than the analysis that considers all scenarios.

		Analysis Time		WCET Bound	
Program	Size	Local	Global	Local	Global
edn	2372	9 s	64 s	149,631	152,818
prime	292	1 s	1 s	5,451	5,996
dry	2624	3 s	110 s	20,202,828	22,248,372

Table 8.1: Running times and WCET with and without local worst-case

The three programs in this experiment contain a collection of real-time algorithms (filters, CRC computation) in the **edn** executable, the computation of the first 20 prime numbers (**prime**) and the Dhrystone benchmark (**dry**).

The high analysis times are due to the fact that no extra information was provided to the analysis, even loop bound were found by the analysis itself. Also, no tuning in terms of context definitions were made, cf. Chapter 6. The influence of these context settings on the analysis times and results with local and global worst-case analysis are depicted in Figure 8.1 for the Dhrystone benchmark. The times are given against the number of different context to consider for one program point.

The difference of from 22.46% for context bound 1 and 8.03% for the context bounds greater than 7 for the computed WCET for the local worst-case only analysis w.r.t. the global worst-case analysis is not solely because of timing anomalies. Rather, there is a large influence of the timing abstraction we use. Namely, WCETs are computed for each basic block by taking the length of the maximal execution sequence of the instructions in the block as WCET for that block. Consider that two subsequent blocks are executed and that there are two paths going through both blocks. On the first path, the first block takes 42 cycles to execute, the second 24. On the second path, the first block takes 32 cycles and the second one 42 cycles. The WCETs for the basic blocks will be 42 cycles for both blocks, as this is the maximum of the execution times for the blocks. Assume now

[1]As local worst-cases lead to global worst-cases *most* of the time but not *all* the time.

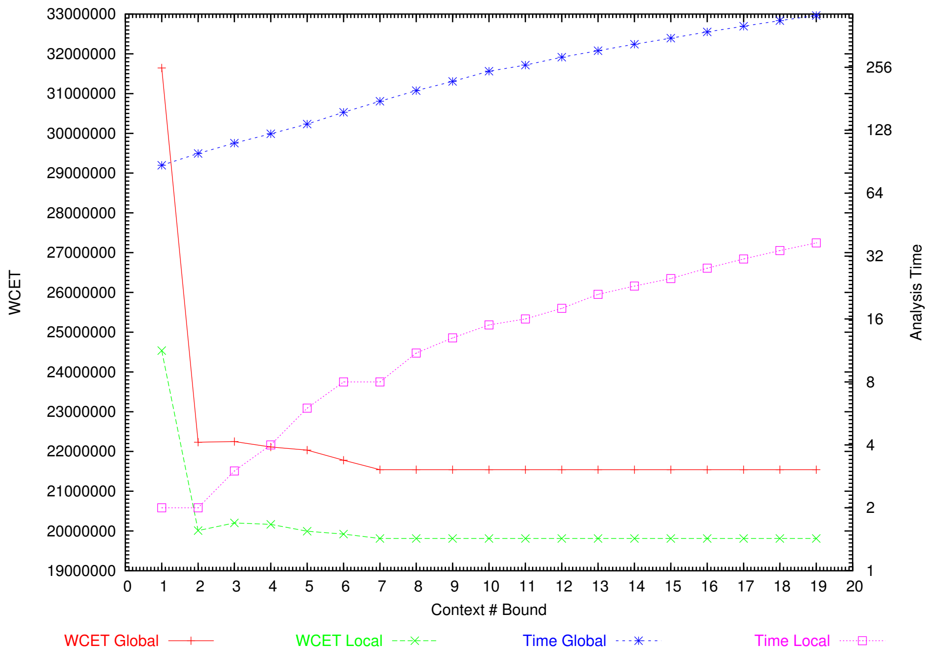


Figure 8.1: WCET and analysis time for local and global worst-case settings

that the first path originates from a local best-case that is nonetheless considered by the global analysis. This path is not considered for the local worst-case only analysis and thus the analysis computes the WCETs 32 and 42 cycles for the first and second block resp. Thus, the total WCET computed for the sequence of the two blocks is lower (32+42 cycles) than the one computed by the global analysis (42+42 cycles). For the same reason, the bound decreases if more contexts are separated by the analysis because then the number of paths through the same basic block/context pair is reduced.

This experiment also shows that computing global worst-case results is much more costly than the local worst-case analysis. The ratio of the analysis times approaches the number 12.24 for the higher number of contexts for this small example.

Another point that is always present when using abstract interpretation as the basis for an analysis is the influence of the abstractions and approximations. An abstracted element is not capable of holding the same amount of information as the concrete one, e. g. for the abstract cache components we only have information about a subset of the memory blocks that are guaranteed to be in the cache when program execution reaches a given point. Also, joins in the CFG lead to the merging of the information at the incoming edges. For correctness reasons the merge must be at least as unprecise as the least upper bound defined on the lattice. Furthermore, the information is lost, on which path the information was computed, which can lead to quite imprecise abstract information. In Figure 8.2 we have two joins in the CFG, after the `if` statements. All paths through this CFG are disjoint in the sense that any execution that goes along the `then` branch of the first `if` goes along the `else` branch of the second `if` statement. And all executions along the `else` branch of the first `if` go along the `then` branch of the second `if`.

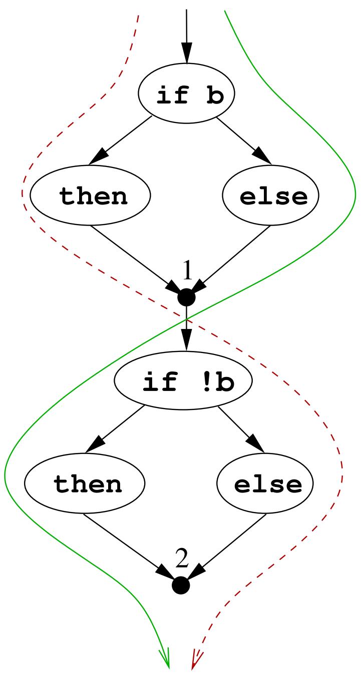


Figure 8.2: Two joins with information loss

So the only paths through the CFG are the ones depicted in the figure. A naive analysis is not able to consider only these two paths through the CFG. Rather, at point **1** in the figure it merges the information of both branches of the first `if` and looses the information along which path of the `if` this point was reached. Then it cannot be decided that information should only be propagated along the reverse branch of the second `if`. For cache analysis, e. g., this means that if the alternating branches of the two `if` statements access the same data, these accesses cannot be classified as at the point `1` it cannot be guaranteed that the data is in the cache for *both* branches of the first `if`.

A similar situation exists for loops. Here, a naive analysis is not able to separate the first loop iteration from the second or third, etc. This is especially bad for cache analysis as the first iteration of a loop normally loads most of the necessary instructions of the loop into the cache and subsequent iterations find the data in the cache. The naive analysis would classify the accesses to the first instructions in a cache line in a loop as cache misses. It has already been mentioned in Section 3.3.1 how this can be avoided by using *contexts* for the data-flow analysis. The contexts make different paths through the CFG distinguishable by the DFA. The same principle can, if one desires, be applied to the situation of alternating `if` branches mentioned above. Introducing more contexts does naturally increase at first the costs of the analysis as more data flow elements must be propagated. However, it may reduce the number of iterations necessary for one node in the CFG, because the information stabilizes earlier. Finding a good compromise is a matter of tuning the parameters that govern the generation of different context for the analysis.

Other aspects related to the analysis technique is the reduced complexity of the overall analysis if it is split into separate analyses running after each other. It has already been discussed in Section 6.3 that for the case of separating the cache from the pipeline analysis the loss in precision can be severe due to necessary conservative approximations for the interdependent analyses.

Another point is that the hardware itself may have a behavior that makes a static analysis difficult. This is the case if we cannot gather precise information about the behavior of a component during execution of a program block if we don't have any information about the component at the beginning of the block. The non-existence of information about a component is common for static analysis if program blocks are analyzed in isolation (modular analysis) as the contexts by which the block can be reached are unknown and must be approximated. Furthermore, joins in the CFG may lead to loss of information about components, because the static analysis computes information that is valid for *all* executions passing a given program point. Therefore, if control-flow joins only the information guaranteed to be correct for all incoming edges can survive in the analysis. Concrete examples of such hardware are the caches of the Motorola ColdFire 5307 and the PowerPC 755. As mentioned in Section 2.5.2, there is only one global replacement counter c for the cache. If an analysis does not have information about the value of this counter[2], then it can be shown that no information about the counter can ever be obtained that is different from $0 \leq c \leq 3$ by a *first-order analysis*. Such an analysis does not take into account the *path* by which the current program point was reached but only the point (instruction) before the current one. A higher order analysis would naturally be much more complex. For the ColdFire cache the absence of knowledge about the replacement counter has the effect that it can only be guaranteed statically that the last accessed cache block is in the cache. Furthermore, the absence of blocks cannot be guaranteed (may-analysis) as blocks can never be shown to be replaced from the cache. In practice, this virtually reduces the cache size of the MCF 5307 to 1/4 of the real cache for the analysis.

The pseudo LRU cache of the MPC755 shows a similar effect as the replacement strategy is not monotone but "jumps" due to the decision tree used to determine the block to be replaced next, cf. Section 2.6.4. In [HLTW03] the loss in precision of the analysis of the pseudo LRU cache of the MPC755 compared to a real 8-way associative LRU cache was investigated. Figure 8.3 shows the results of one experiment. The experiment consisted in analyzing one executable with different analysis settings for the contexts considered[3].

The measure chosen for comparison was the number of lines guaranteed in the cache (i. e. must analysis was performed) summed over all program points and contexts and divided by the number of contexts and program points. The example program was a 84kB PowerPC code that contains code pieces typical for avionics (filters, CRC computations, etc). The ratio of the LRU and PLRU results is up to 1.609. As the whole program is just 2.5 times larger than the PowerPC cache of 32kB, this result already shows a considerable loss of information due to the bad analyzability of the PLRU strategy. Also, this shows that a LRU strategy recovers from unknown information after a sequence of accesses. Here, the cache contents and ages of the blocks in the cache are known quite precise. This is because the LRU replacement is more "monotone" in the aging of blocks in the cache.

Another source for imprecision in the analysis related to the hardware may arise if the the average (or best) case and the worst-case behavior of components differ significantly. One example are again caches. Another example are "shortcuts" in the instruction execution. The MPC755 for example can shorten integer multiplications if the upper $8 * n$, $1 \leq n \leq 4$, bits of one operand are zero. Instead of three cycles, a multiplication may then only take one cycle. While this increases average-case performance a bit, it makes the analysis more imprecise, as it is mostly not possible to obtain tight intervals for the operands of multiplica-

[2]So it must assume it has values $0 \leq c \leq 3$.

[3]"vivu" in the figure refers to the virtual unrolling of loops mentioned above, while "cs0" is the callstring approach with length 0, cf. [Mar99a, Mar99b].

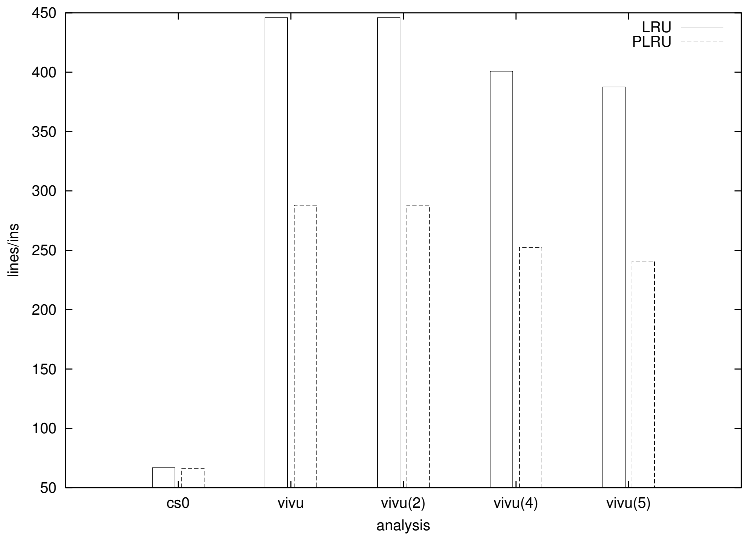


Figure 8.3: PLRU vs. LRU cache analysis

tions without increasing the analysis costs considerably. Thus, all cases of execution time for the multiplication must be considered by the analysis (1, 2, 3 cycles), increasing the costs and decreasing the precision, as few fast multiplications can be predicted.

Last, the structure of the program code to analyze has a great influence on the analysis costs and the precision of the results. If the code is generated from higher-level control laws, it is quite easy to analyze, especially if no optimizing compilers have been used to compile the generated C code to object code. Hand written or highly optimized code makes the analysis more costly, as it is less likely to obtain good information that can limit the number of successor pipeline states that have to be considered. Also, the quality of the generated bounds may be poor, as the analysis may not be able to separate enough context in the optimized code, cf. above.

8.2 Advice for Predictable Systems

In the following we list a number of processor properties whose combination will allow high precision in statements about the timing behavior and a modular design of the timing analyzer.

- Separate data and instruction caches: separate caches eliminate the interdependencies of instruction prefetching and data accesses. This way the precision loss of separate cache and pipeline analyses can be reduced. In an integrated analysis, worst case assumptions can be made more easily since these dependencies do not have to be considered.

- Cache replacement strategies: these should be immune against “chaos”. This means that when cache contents are not known at one point, subsequent accesses can recover knowledge about the new cache contents. The ColdFire cache with its global replacement counter does not allow to recover knowledge about the counter, if one has no information on its value at some point. LRU replacement strategies recover from “chaos”: after some cache updates, the ages of the new elements in the cache are known.

  Naturally, the update strategy should be (locally) deterministic, otherwise little can be statically said about cache contents.

  The cache architecture should allow both must and may analyses for the caches. Neither the ColdFire nor the PowerPC 755 cache make this possible, although this information about what is guaranteed *not* to be in the cache, is valuable in restricting the non-determinism in the pipeline analysis.

- Branch prediction if any should be static: the modeling of dynamic branch prediction would lead to an even more complex integrated analysis. A static, separate and precise analysis of dynamic branch prediction is difficult since it also depends on the pipeline state.

- Out-of-order execution should be limited: with out-of-order execution, one has to consider the effects of all possible interleavings of instructions. Clearly, this is difficult and imprecise to do statically in a separate analysis since there are many possible interleavings, whereas a worst-case interleaving is not likely to occur during execution but must be assumed to ensure a correct result. In an integrated approach all interleavings are considered but most of them will not be worst cases. The required granularity of the pipeline model for this both increases design complexity and analysis complexity.

- Shortcuts should be avoided: in general, shortcuts in the hardware design, e. g. special cases to accelerate some operation, if certain (dynamic) conditions hold, should not be used. While they definitely improve average performance, they have little gain in running typical real-time tasks; nonetheless, they must be modeled in quite some detail in the pipeline analysis or give raise to increased nondeterminism.

Chapter 9

Conclusions

In summary, the idea is to give all of the information to help others to judge the value of your contribution; not just the information that leads to judgement in one particular direction or another.
Richard P. Feynman

Hard real-time systems must be verified, requiring, among others, knowledge about upper bounds for the worst-case execution time (WCET) of the tasks in the system. Static analysis working on the program code is currently viewed as the only safe and practical method of obtaining WCET bounds.

Features of modern hardware (cache, pipelines, SDRAM, ...) which have been introduced to improve average-case performance of systems are not easy to handle within static analysis due to their history sensitive behavior and strong interactions between different features (e. g. caching and branch prediction).

Precise results can be difficult to obtain, because their exist bad worst-case scenarios for the performance of certain features which have to be assumed for a safe analysis result in case the scenario cannot be ruled out by the analysis. Furthermore, the interaction of features may make it difficult or impossible to decide *locally* which of the possible behaviors will lead to the *globally* longest execution time (timing anomalies). This necessitates an integrated, global analysis approach in order to conservatively capture all possible effects on the execution time.

This work presents a new approach to the problem of obtaining safe bound for WCETs for complex hardware architectures and realistically sized programs. Our approach is semantics based, utilizing safe abstractions using the technique of *abstract interpretation*. This way, the correctness of the results can be proven, another advantage over currently advocated methods. We use a model of the system, containing encapsulated *units* with inner state and update rules, communicating via typed *signals*. The model forms a cycle-precise simulation of program execution in the hard real-time system. Components of this model are then abstracted using the framework of abstract interpretation, resulting in an *abstract model* that still can be simulated in a cycle-precise way.

Finding all possible simulation sequences starting from a set of system start states in this nondeterministic abstract model allows to obtain a WCET bound for the basic blocks of the program and successor states to the subsequent basic blocks of the program. Integration of this technique into a data-flow analysis over the control-flow graph of the program gives an analysis that delivers WCET for all the basic blocks of a program.

In this thesis, a concrete semantics for the definition of cycle update programs in a model has been given together with a semantics describing the cycle-wise execution. A method to define abstractions for this concrete semantics has been presented and its correctness has been proven. The correctness proof can be applied to other forms of semantics for cycle updates as well. The approach has been exemplified by giving two models for the Motorola ColdFire 5307 and the PowerPC 755.

A complete WCET tool, **aiT**, has been implemented with this analysis as the central component. The other phases of the tool are also described in this thesis.

9.1 Predicting Modern Hardware

Timing anomalies and bad worst-case behavior make the precise, safe and fast analysis of modern hardware difficult. As widely-used processors are normally designed towards a good average-case performance, little attention is paid to the worst-case scenarios from the side of chip manufacturers. For hard real-time systems, the average-case performance is of less importance than the worst-case performance, since only the latter can be assumed to be available in the scheduling analysis.

Bad worst-case performance can be defined in terms of the difference between average and worst-case behavior. The smaller the gap between both cases, the better the worst-case performance. If one behavior has a fast best or average-case, e. g. a cache hit and a slow worst-case, e. g. a cache miss, then an analysis must either be very detailed, e. g. the must and may analyses for caches must be performed as part of the analysis, or it will be in danger of delivering results that are too far away from the real worst-case, e. g. by assuming that every memory access is a cache miss. In the latter case, the analysis is practically useless, in the former it will be more complex and costly.

As WCET analysis has to consider all behaviors of the system if timing anomalies cannot be ruled out[1], it is most efficient if there are not many possible behaviors for a system component. E.g. static RAM has a fixed access timing, while accesses to SDRAM take different amounts of time, depending on the pages open in the memory banks accessed. An analysis of SDRAM has to consider both page hit and miss behaviors. This, too, makes the analysis more complex and decreases the speed of the analysis.

Even if one is willing to use a more complex analysis, e. g. to analyze cache behavior more precisely, it may be difficult to obtain a precise static analysis. This is because a static analysis has to correctly handle the case that nothing is known about the component, e. g. the cache contents are unknown at the beginning of the (modular) analysis of a procedure in the program. It depends on the system component, whether an (first-order) analysis is able to compute precise information while analyzing the procedure. E.g. for a LRU cache, cache analysis can compute information for all the lines in the cache, as old content (and thus, the *unknown* content) of the cache is guaranteed to be replaced with new content. The cache of the Motorola ColdFire 5307 with its pseudo round-robin replacement does *not* allow to compute information for the whole cache, as the cache replacement counter value remains unknown if it is unknown at the beginning of the procedure, cf. Section 2.5.2.

9.2 Practical Usability

The methods and observations of this thesis have been tested by implementing a WCET tool, **aiT**, for two processors, the Motorola ColdFire 5307 and the PowerPC 755. This work was done during the DAEDALUS project, which aimed at the validation of critical software by static analysis and abstract testing. Airbus France was the main industrial partner in the project and the end user for the tools and new methodologies developed during the project. Consequently, **aiT** was developed and tested in the setting of analyzing avionics software.

As the results of the project are considered for application in the verification toolset for the Airbus project, an extensive evaluation of all tools, including **aiT**, has been performed by Airbus France. This evaluation was performed solely by people from the software validation group of Airbus France, choosing their own benchmarks. This ensured an unbiased evaluation process as neither the benchmarks, nor the settings of parameters of the analysis process, etc, could be chosen by the developers of **aiT** to direct the results into the “desired” direction.

[1]The problem of ruling out timing anomalies is difficult because all possible interactions between components have to be considered.

In the final report on the DAEDALUS project, w.r.t. this evaluation process, the EU review concluded:

> The assessment performed by the end user [Airbus] ... was considered of exceptional quality.

A small part of the evaluation results have been published in [TSH+03]. Due to the sensitive nature of the software involved (flight control), the main parts are not disclosed to the public. The reviewers of the project, who had been presented the results during the review, summarize the overall results of **aiT**'s evaluation as follows in the final review report:

> The results obtained on this topic [WCET analysis], and the improvements of the ABSINT tool is one of the most important results of the Daedalus project, from both technical and industrial points of view. The WCET analysis tool for the ColdFire processor has been very successful. The final evaluation of the tool for PowerPC was expected to be at hand about a month after the final review. The ABSINT tool is probably the best of its kind in the world, and it is justified to consider this result as a breakthrough.

After the DAEDALUS project, the development of **aiT** for the PowerPC 755 has been continued. In this course, work has been started to model the system controller of the target hardware processor board. Validation and introduction of this specialized **aiT** version is currently under way.

Beside the PowerPC 755 version, **aiT** is available as a commercial tool marketed by AbsInt GmbH, for a number of other processors, e. g. PowerPC 5xx and ARM, cf. `http://www.absint.com/ait`.

In summary, our approach can be applied to real-life problems in the area of avionics with success. It is the first one to handle a combination of advanced processor feature like

- speculative execution,
- prefetching,
- branch prediction,
- out-of-order execution,
- and caching

together in a provably correct way for a real-life processor (independently) evaluated on real-life benchmarks. This is also the first time that an industrial strength commercial tool for complex processors has been designed and is being marketed.

9.3 Future Work

The application to areas other than avionics in which hard real-time systems have to be analyzed, e. g. automotive industry or train control systems, is currently under investigation in other projects, e. g. the AVACS Transregional Collaborative Research Center 14 sponsored by the German Science Foundation: `http://www.avacs.org`.

The design of processor models based on handbooks, experiments and information obtained from the processor designers is a complex and lengthy process. The process may introduce errors at the level of the model. As the correctness of the model can only be verified with test runs of the analyzer and comparison against real execution traces, these errors may be costly to find. As an alternative, models can be made starting from an authoritative source, namely the Verilog or VHDL descriptions of the processor (and the other system hardware). The modularization and signal concept is similar to the one used in this thesis. The abstraction methodologies can be translated to these languages in a similar way, at least if the models are clocked, i. e. operations are synchronized to the processor clock edge. During AVACS, this approach will be studied for the (publically available) LEON2 SPARC clone.

Another area of active research is to reduce the costs of the WCET analysis. One aspect is to reduce the size of the state information that has to be kept in the abstract model. Another aspect is to reduce the degree of nondeterminism in the analysis, resulting in less successor states for one abstract pipeline state during the analysis. If a processor does not possess timing anomalies, then only the local worst-case has to be considered in the case of non-determinism. We are investigating approaches to verify the absence of timing anomalies by using model checking on processor models. With this, one can either show the general absence of anomalies for a processor or the absence of anomalies during the execution of a fixed program. Model checking is also usable to determine that one component of a pipeline state subsumes another, i. e. the information in one component uniquely (w.r.t. the abstractions made) defines the contents of the other. Then, the second component can be safely removed from the state, reducing the complexity of the WCET analysis.

Bibliography

[aiS] http://www.aisee.com. *aiSee Home Page*.

[AK95] D. P. Appenzeller and A. Kuehlmann. Formal Verification of a PowerPC Microprocessor. In *Proceedings of the IEEE International Conference on Computer Design (ICCD '95)*, Austin, Texas, 1995.

[Ash02] P. J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, Academic Press, 2nd edition, 2002.

[BBB03] A. Burns, G. Bernat, and I. Broster. A Probabilistic Framework for Schedulability Analysis. In R. Alur and I. Lee, editors, *Proceedings of the Third International Embedded Software Conference, EMSOFT*, volume 2855 of *Lecture Notes in Computer Science*, pages 1–15, 2003.

[BBCZ98] S. Berezin, A. Biere, E. Clarke, and Y. Zhu. Combining Symbolic Model Checking with Uninterpreted Functions for Out-of-order Processor Verification. In *Proceedings of the Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98*, volume 1522 of *Lecture Notes in Computer Science*, Palo Alto, California, USA, 1998. Springer.

[BCM$^+$92] J. Burch, E. Clark, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2), 1992.

[BCP02] G. Bernat, A. Colin, and S. M. Petters. WCET Analysis of Probabilistic Hard Real-Time Systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, Austin, Texas, USA, December 2002.

[BD94] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Conference on*

Computer-Aided Verification, volume 818 of *Lecture Notes in Computer Science*, pages 68–80. Springer-Verlag, 1994. Stanford, California, June 21–23, 1994.

[Ber] G. Berry. The Esterel V5 Language Primer. Esterel Technologies.

[BHE91] D. G. Bradlee, R. R. Henry, and S. J. Eggers. The Marion System for Retargetable Instruction Scheduling. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240, Toronto, ON, Canada, June 1991. ACM Press.

[BN94] S. Basumallick and K. Nilsen. Cache Issues in Real-Time Systems. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, June 1994.

[Bur96] J. R. Burch. Techniques for Verifying Superscalar Microprocessors. In *Proceedings of the 33rd Design Automation Conference DAC 96*, pages 552–557, Las Vegas, USA, June 1996. ACM Press.

[BW90] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages*. Addison Wesley, 1990.

[CC77] P. Cousot and R. Cousot. A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, January 1977.

[CC79] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM Symposium on Principles of Programming Languages*, 1979.

[CC91] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *BIGRE*, 74:107–110, October 1991.

[CC92a] P. Cousot and R. Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In

M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP'92,*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295, Leuven, Belgium, August 1992. Springer.

[CCKT86] D. Callahan, K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Constant Propagation. *ACM SIGPLAN Notices*, 21(7):152–161, June 1986. Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, USA.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles Of Programming Languages*, pages 84–96. ACM Press, 1978.

[Cou81] P. Cousot. Semantic Foundations of Program Analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[CP00] A. Colin and I. Puaut. Worst Case Execution Time Analysis for a Processor with Branch Prediction. *Real-Time Systems, Special issue on worst-case execution time analysis*, 18(2):249–274, April 2000.

[CP01] A. Colin and I. Puaut. A Modular and Retargetable Framework for Tree-based WCET Analysis. In *13th Euromicro Conference on Real-Time Systems*, pages 37–44, June 2001.

[DO192] European Organisation for Civil Aviation Electronics. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Dec 1992.

[DP97] W. Damm and A. Pnueli. Verifying out-of-order executions. In *Proceedings of the IFIP WG 10.5 International Conference on Correct Hardware Design and Verification Methods*, pages 23–47. Chapman & Hall, Ltd., 1997.

[Eng02] J. Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Science and Technology, Uppsala University, 2002.

[Erm03] A. Ermedahl. *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. PhD thesis, Faculty of Science and Technology, Uppsala University, 2003.

[Fer97] C. 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 the first International Workshop on Embedded Software, EMSOFT 2001*, volume 2211, 2001.

[FKL+99] C. Ferdinand, D. Kästner, M. Langenbach, F. Martin, M. Schmidt, J. Schneider, H. Theiling, S. Thesing, and R. Wilhelm. Run-Time Guarantees for Real-Time Systems — The USES Approach. In *Proceedings of Informatik '99 – Arbeitstagung Programmiersprachen*, Paderborn, 1999.

[GHDN99] P. Grun, A. Halambi, N. Dutt, and A. Nicolau. RTGEN: An Algorithm for Automatic Generation of Reservation Tables from Architectural Descriptions. In *Proceedings on the 12th International Symposium on Systems Synthesis*, 1999.

[Gra69] R.L. Graham. Bounds on Multiprocessing Timing Anomalies. *SIAM Journal of Applied Mathematics*, 17(2):416–429, March 1969.

[HAM+99] C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon. Bounding Pipeline and Instruction Cache Performance. *IEEE Transactions on Computers*, 48(1):53–70, January 1999.

[HBL+95] Y. Hur, Y. H. Bae, S. Lim, S. Kim, B. Rhee, S. L. Min, C. Y. Park, M. Lee, H. Shin, and C. S. Kim. Worst Case Timing Analysis of RISC Processors: R3000/R3100 Case Study. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 308–319, December 1995.

[HGG+99] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: A Language for Architecture Exploration through

Compiler/Simulator Retargetability. In *Proceedings of the Design, Automation and Test in Europe Conference, DATE'99*, pages 485–490, March 1999.

[HLTW03] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7), July 2003.

[HP96] J. L. Hennessy and D. A. Patterson. *Computer Architecture. A Quantitative Approach.* Morgan Kaufmann Publishers, 2nd edition, 1996.

[HQR98] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You Assume, We Guarantee: Methodology and Case Studies. In *Proceedings of the 10th International Conference on Computer-aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer-Verlag, 1998.

[HT01] R. Heckmann and S. Thesing. Cache and Pipeline Analysis for the Coldfire 5307. Technical report, Saarland University, 2001.

[HWH95] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 288–297, December 1995.

[HYHD95] R. C. Ho, C. Han Yang, M. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 404–413. Association for Computing Machinery, June 1995. Santa Margherita Ligure, Italy, 22-24 June 1995.

[Hym02] C. Hymans. Checking Safety Properties of Behavioral VHDL Descriptions by Abstract Interpretation. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 444–460. Springer, 2002.

[Hym03] C. Hymans. Design and Implementation of an Abstract Interpreter for VHDL. In *12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME'03)*, Lecture Notes in Computer Science. Springer, 2003.

[Int91] Intel Corporation. *i960 KA/KB Microprocessor Programmer's Reference Manual*, 1991.

[Int99] Intel Corporation. *PC SDRAM Specification*, November 1999. Revision 1.7.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The BCS Computer Journal*, 29(5):390–395, January 1986.

[JPM02] S. Jolly, A. Parashkevov, and T. McDougall. Automated Equivalence Checking of Switch Level Circuits. In *Proceedings of the 39th Design Automation Conference, DAC 2002*, New Orleans, USA, 2002. ACM.

[JSD98] R. B. Jones, J. U. Skakkebaek, and D. L. Dill. Reducing manual abstraction in formal verification of out-of-order execution. In *Formal Methods in Computer-Aided Design*, pages 2–17, 1998.

[KT99] D. Kästner and S. Thesing. Cache Aware Pre-runtime Scheduling. *Real-Time Systems Journal*, 17(2/3):235–250, November 1999.

[Lan98] M. Langenbach. CRL – A Uniform Representation for Control Flow. Technical report, Saarland University, Faculty for Computer Science, 1998.

[LBJ$^+$95] S. Lim, Y. H. Bae, G. T. Jang, B. Rhee, S. L. Min, C. Y. Park, H. Shin, K. Park, and C. S. Kim. An Accurate Worst Case Timing Analysis Technique for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), July 1995.

[LHYM$^+$96] C.-G. Lee, J. Hahn, Seo Y.-M., S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling. In *Proceedings of the 16th Real-Time Systems Symposium*, 1996.

[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*, 20:46–61, 1973.

[LMA97] Y.-T. S. Li, S. Malik, and A.Wolfe. Cache Modeling for Real-Time Software: Beyond Direct Mapped Instruction Caches. *IEEE Real-Time Systems Symposium*, January 1997.

[LRM+94] S. Lim, B. Rhee, S. L. Min, C. Y. Park, H. Shin, and C. S. Kim. Issues of Advanced Architectural Features in the Design of a Timing Tool. In *Proceedings of the 11th IEEE Workshop on Real-time Operating Systems and Software*, 1994.

[LS98] T. Lundqvist and P. Stenström. Integrating Path and Timing Analysis Using Instruction-Level Simulation Techniques. In F. Mueller and A. Bestavros, editors, *Proceedings of the ACM SIGPLAN Workshop Languages, Compilers and Tools for Embedded Systems (LCTES)*, volume 1474 of *Lecture Notes in Computer Science*, pages 1–15, 1998.

[LS99] T. Lundqvist and P. Stenström. An Integrated Path and Timing Analysis Method based on Cycle-Level Symbolic Execution. *Real-Time Systems Journal*, 17(2/3):183–207, November 1999.

[Lun02] T. Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories*. PhD thesis, Dept. of Computer Engineering, Chalmers University of Technology, Sweden, June 2002.

[Mar98] F. Martin. PAG – an Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer*, 2(1), 1998.

[Mar99a] F. Martin. Experimental Comparison of call string and functional Approaches to Interprocedural Analysis. In S. Jaehnichen, editor, *Proceedings of the 8th International Conference on Compiler Construction*, volume 1575 of *Lecture Notes in Computer Science*, pages 63–75. Springer, 1999.

[Mar99b] F. Martin. *Generation of Program Analyzers*. PhD thesis, Saarland University, 1999.

[McM93] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[McM98] K. McMillan. Verification of an Implementation of Tomasulo's Algorithm by Compositional Model Checking. *Lecture Notes in Computer Science*, 1427, 1998.

[Met04] A. Metzner. Why Model Checking Can Improve WCET Analysis. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV'2004*. Springer, July 2004.

[Mic92] Sun Microsystems. The SuperSPARC Microprocessor. Technical White Paper, May 1992.

[Mic03] Micron Technology, Inc. *64MB Synchronous DRAM MT48LC16M4A2 Datasheet*, 2003.

[Moo65] G. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[Mot00] Motorola. *MCF5307 ColdFire Integrated Microprocessor User's Manual*, August 2000. MCF5307UM/D, Rev. 2.0.

[MTH$^+$02] P. Mishra, H. Tomiyama, A. Halambi, P. Grun, N. Dutt, and A. Nicolau. Automatic Modeling and Validation of Pipeline Specifications Driven by an Architecture Description Language. In *Proceedings of ASPDAC-2002/VLSI Design 2002*, 2002.

[MWH94] F. Mueller, D.B. Whalley, and M. Harmon. Predicting Instruction Cache Behavior. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compiler and Tool Support for Real-Time Systems*, 1994.

[NN94] K. Narasimhan and K. Nilsen. Portable Execution Time Analysis for RISC Processors. In *ACM PLDI Workshop on Language, Compiler, and Tool Support for Real-Time Systems*, June 1994.

[NNH99] F. Nielsen, H. Nielsen, and C. Hankin. *Principles of Program Analysis*. Addison-Wesley, 1999.

[NR95] K. Nilsen and B. Rygg. Worst-Case Execution Time Analysis on Modern Processors. In *Proceedings of the Second ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Real-Time Systems, LCT-RTS*, June 1995.

[Ope02] Open SystemC Initiative. *SystemC User's Guide*, 2.0 edition, 2002.

[PPC97a] Motorola. *MPC750 RISC Processors User's Manual*, 1997.

[PPC97b] Motorola. *PowerPC Microprocessor Family: The Programming Environments for 32-bit Microprocessors*, 1997.

[PZHM98] S. Pees, V. Zivojnovici, A. Hoffmann, and H. Meyr. Retargetable Timed Instruction Set Simulation of Pipelined Processor Architectures. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, September 1998.

[RPTW04] A. Rakib, O. Parshin, S. Thesing, and R. Wilhelm. Component-wise Instruction Cache Behavior Prediction. In *Proceedings of the 4th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Catania, Sicily, Italy, June 2004.

[SA95] T. Shanley and D. Anderson. *PCI System Architecture*. MindShare, Inc., third edition, 1995.

[Sch02] J. Schneider. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2002.

[SF99] J. Schneider and C. Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *Workshop on Languages, Compilers, and Tools for Embedded Systems*, volume 34 of *ACM SIGPLAN Notices*, pages 35–44, May 1999.

[Sha89] A. C. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[Sic97] M. Sicks. Adressbestimmung zur Vorhersage des Verhaltens von Daten-Caches. Master's thesis, Saarland University, 1997.

[Sis98] C. Siska. A Processor Description Language Supporting Retargetable Multi-Pipeline DSP Program Development Tools. In *Proceedings of the 11th International Symposium on System Synthesis*, 1998.

[SJD98] Jens U. Skakkebæk, Robert B. Jones, and David L. Dill. Formal Verification of Out-of-order Execution Using Incremental Flushing. In *Proceedings of the 10th International Conference on Computer Aided Verification, CAV'98*, June 1998.

[SP81] M. Sharir and A. Pnueli. Two Approaches to Interprocedural Data Flow Analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Application*, chapter 7, pages 189–233. Prentice-Hall, 1981.

[Sto03] J. Stokes. Understanding Moore's Law. WWW, February 2003. http://www.arstechnica.com/paedia/m/moore/moore-1.html.

[TBW94] K. Tindell, A. Burns, and A. Wellings. An Extensible Approach for Analyzing Fixed Priority Hard Real-Time Tasks. *The Journal of Real-Time Systems*, 6(2):133–151, March 1994.

[The02] H. Theiling. *Control Flow Graphs for Real-Time System Analysis - Reconstruction from Binary Executables and Usage in ILP-based Path Analysis*. PhD thesis, Saarland University, 2002.

[Tho64] J. E. Thornton. Parallel Operation in the Control Data 6600. In *Proc. Fall Joint Computer Conference*, pages 33–40, 1964.

[TM91] D.E. Thomas and P.R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, Massechusetts, 1991.

[TMLA97] S. Thesing, F. Martin, O. Lauer, and M. Alt. *PAG: User's Manual*. Saarland University, 1.0 edition, 1997.

[Tom67] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM J. Research and Development*, 11(1):25–33, January 1967.

[TSH+03] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics. In *Proceedings of the International Performance and Dependability Symposium (IPDS)*, June 2003.

[Tuo02] I. Tuomi. The Lives and Death of Moore's Law. *First Monday WWW Journal*, 7(11), November 2002. http://firstmonday.org.

[VHD00] Institute of Electrical and Electronic Engineers, New York. *Draft IEEE Standard P1076 2000/D3 VHDL Language Reference Manual*, 2000.

[Wil04] R. Wilhelm. Why AI + ILP is Good for WCET, but MC is Not, nor ILP Alone. In *Proceedings of the Fifth International Conference on Verification, Model Checking and Abstract Interpretation*, Venice, Italy, January 2004.

[WM95] R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

[ZPS+96] V. Zivojnovic, S. Pees, C. Schläger, M. Willems, R. Schoenen, and H. Meyr. LISA – Machine Description Language and Generic Machine Model. In *Proceedings of the International Conference on Signal Processing Applications and Technology*, October 1996.