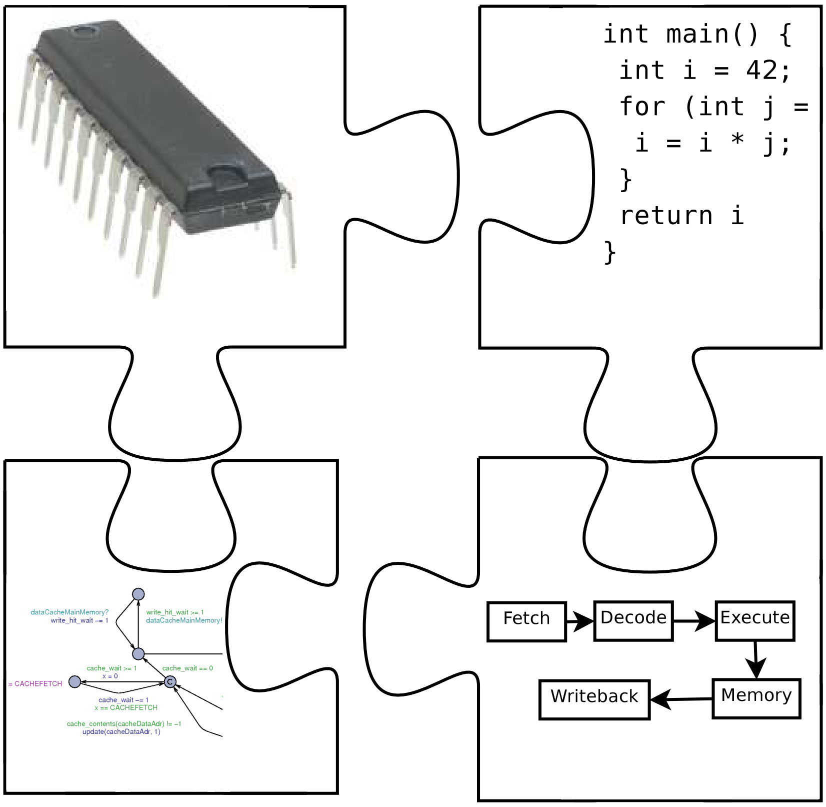
METAMOC

Modular Execution Time Analysis using Model Checking



```
int main() {
 int i = 42;
 for (int j =
  i = i * j;
 }
 return i
}
```

Master's thesis by

Andreas Engelbredt Dalsgaard
Mads Christian Olesen
Martin Toft

June 2, 2009

Title:
Modular Execution Time Analysis using Model Checking (METAMOC)

Project period:
DAT6, the 2nd of February to the 2nd of June, 2009

Project group:
Computer Science, d601a

Members:
Andreas Engelbredt Dalsgaard
Mads Christian Olesen
Martin Toft

Supervisors:
Kim Guldstrand Larsen
René Rydhof Hansen

Copies:
6

Pages in the report:
100

Pages including appendices:
105

Abstract:

The ability to determine safe and sharp worst-case execution time (WCET) for processes is very important when scheduling real-time systems, as it influences the reliability and efficiency of the resulting systems. This thesis presents METAMOC, a flexible WCET analysis method based on model checking and static analysis that determines safe and sharp WCETs for processes running on hardware platforms featuring caching and pipelining.

The method is divided into four loosely coupled sub-analyses. To demonstrate and evaluate the method, it is implemented for the ARM920T processor and tested successfully on most of the WCET benchmark programs from Mälardalen Real-Time Research Center. The flexibilty of the method allows for easy replacement and reuse of parts of the implementation, in order to add support for additional hardware platforms.

Experiments with the implementation show that the principles used in the method work very well, and that taking especially caching into account is worth the effort.

Preface

The present report is our master's thesis, which documents the results of our work during the spring semester of 2009. The thesis is on the topic of formal systems and marks the completion of a specialisation year in the Distributed and Embedded Systems (DES) research unit at the Department of Computer Science at Aalborg University, Denmark.

The presented results build upon preliminary work documented in the report *Worst-Case Execution Time Analysis for Real-Time Systems* [7], which was written by yours truly during the autumn semester of 2008. Some parts of this thesis are derived from the autumn report, and, even though all the parts have been improved, they deserve to be mentioned. In this thesis, the parts are Sections 2.1–2.3, and Chapter 3.

We have aimed at making the thesis self-contained for fellow students and researchers in the domain of formal systems, which means that we assume basic knowledge of real-time systems, timed automata and verification of properties of timed automata. Furthermore, mathematical skills corresponding to a bachelor level in computer science are advantageous.

In addition to presenting our work in this thesis, we have engaged in a number of research preparatory activities during the specialisation year. They have been a part of the elite student programme in embedded systems offered by Aalborg University, which we enrolled in. On March 12-13 we gave an invited talk on the topic of "cache and pipeline analysis using model checking" at the Danish Network for Intelligent Embedded Systems (DaNES) Mini Case Event [1]. On April 23 we did a presentation at the National Natural Science Foundation of China and Danish National Research Foundation's joint symposium on Information- and Communication Technology. On June 22-24 we will present our work in Aachen, Germany, as our extended abstract [15] for the SSV 2009 Doctoral Symposium [3] has been accepted. The event, which is held during the 4th International Workshop on Systems Software Verification (SSV 09), consists of a number of technical sessions and is dedicated to presentations by Ph.D. students and young, upcoming researchers.

The source code for the implementation of the developed METAMOC method is available at

`http://metamoc.martintoft.dk`

Acknowledgements. We would like to thank our supervisors, professor Kim Guldstrand Larsen and associate professor René Rydhof Hansen, for providing valuable input and references, and for always being able to find time for us in their busy schedules. Additionally, we would like to thank associate professor

Bent Thomsen from the Database and Programming Technologies (DPT) research unit, who was censor on our autumn project exam and in that connection gave a number of valuable pieces of advice on project direction and applicability of the developed method. Finally, we would like to thank F-klubben, the social club in the department, for hosting many enjoyable events that helped us escape from our keyboards once in a while.

Andreas Engelbredt Dalsgaard

Mads Christian Olesen

Martin Toft

Aalborg, June 2, 2009

Contents

Chapter 1

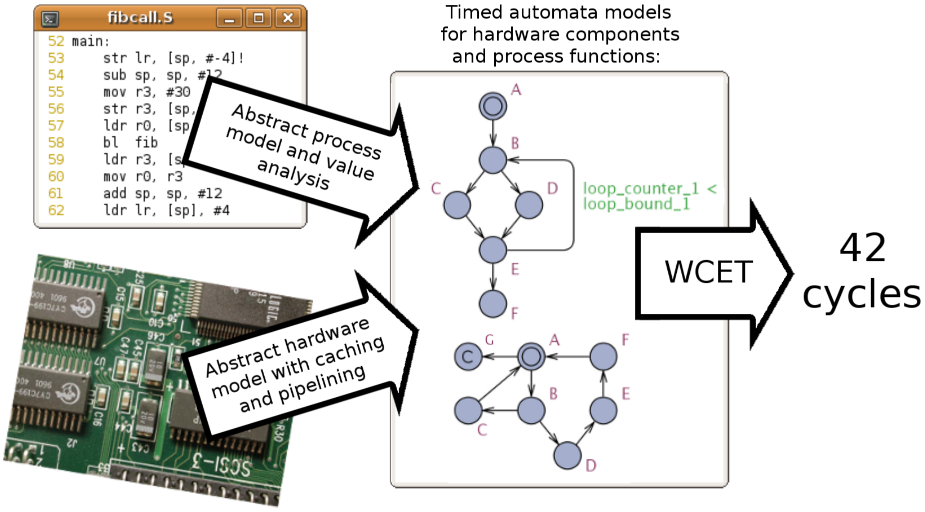
Introduction

Through the use of trains, planes, food production plants, hospital equipment, nuclear reactors, etc., the modern world has become increasingly dependent on advanced technology. The technologies are often constructed as real-time systems (RTSs) and thus contain components in which time plays a significant role. For example, guidance systems in planes and rockets must sample a set of sensors within precise intervals to be able to control the flight actuators correctly. Another example is X-ray machines, which must ensure that patients are only exposed to certain volumes of X-rays during scans for fractures [24]. In both examples, bad timing in software might cause loss of human lives or destruction of expensive equipment, which emphasise why scheduling of RTSs deserve attention.

A RTS is a concurrent software system with a number of processes, each marked with a deadline. The deadline of a process specifies that its execution must be completed within a certain amount of time [13]. RTSs are divided into two classes: soft and hard. Systems in the former class tolerate that processes miss their deadline and respond with lower quality of service, whereas systems in the latter class consider deadline misses to be critical failures [13]. Although most important to hard RTSs, systems in both classes share the need to schedule the execution of processes in a way such that all processes are completed before their deadline, ensuring that the associated physical system behaves as intended.

In order to schedule the processes in a RTS, the worst-case execution time (WCET) of each process in the system is needed [13]. The WCET depends on the hardware platform executing the processes and is determined by performing a WCET analysis. The analysis is a daunting task, since both the behaviour of processes and of the hardware platform executing the processes can be arbitrarily complex. Limiting the behaviour that processes are allowed to exhibit and abstracting away complex hardware components is a common way to obtain usable overestimates of WCETs. The challenge faced by developers of WCET analysis methods is to have very few behaviour limitations and to model the hardware platform as accurately as possible, while still providing a safe yet usable approximation.

Traditionally, research on WCET analysis has focused on ensuring reliable operation of RTSs, while the potentially low efficiency of the resulting systems has been mostly ignored and overcome by using more powerful hardware. Complex hardware features, such as caching and pipelining, have typically been



```
main:
    str lr, [sp, #-4]!
    sub sp, sp, #12
    mov r3, #30
    str r3, [sp,
    ldr r0, [sp
    bl  fib
    ldr r3, [s
    mov r0, r3
    add sp, sp, #12
    ldr lr, [sp], #4
```

Figure 1.1: Overview of the METAMOC WCET analysis method.

abstracted away to simplify the analysis. The method presented in this thesis uses a combination of model checking and static analysis to perform a WCET analysis that takes the effects of caching and pipelining into account. Compared to other WCET analysis methods that do not consider caching and pipelining, this method gives rise to more efficient RTSs, which require less expensive and less energy consuming hardware. It also follows that existing hardware can be reused for bigger systems.

An overview of the Modular Execution Time Analysis using Model Checking (METAMOC) method is provided in Figure 1.1. It takes as input a binary executable of a process, typically obtained by compiling high-level source code, and an abstract model of a hardware platform and determines the WCET of executing the process on the platform. The formalism used for the abstract models is networks of timed automata (NTAs). The method determines the WCET in a number of steps. Initially, a disassembled version of the executable is translated into an abstract process model. Next, value analysis is performed on the assembler code to establish register contents at as many locations in the code as possible. The result of the value analysis is subsequently added to the process model. Finally, model checking is used to carry out an abstract simulation of the interactions between the two models and determine the longest path through the simulation. The time of the longest path is the WCET.

To evaluate the method, we have implemented it for the ARM920T processor from ARM Ltd. [28], which is a typical processor for embedded systems. The implementation is evaluated through a series of experiments that are conducted using realistic WCET analysis benchmark programs from the WCET Analysis Project by Mälardalen Real-Time Research Center (MRTC) [4].

Thesis outline. Chapter 2 presents theory on RTSs, scheduling of RTSs, WCET analysis, UPPAAL-specific model checking, and weighted push-down systems (WPDSs). The theory is a prerequisite for understanding the subsequent chapters. In Chapter 3, optimisation techniques found in modern processors, such as caching and pipelining, are explained in detail. In addition, the chapter explains timing anomalies and abstract representations of the optimisation techniques. Chapter 4 introduces the ARM920T processor, the associated ARM assembly language and the situations in which the processor's pipeline stalls. This is necessary knowledge for modelling the hardware and translating the assembly code to UPPAAL models. In Chapter 5, the METAMOC method is presented in full detail using a concrete implementation for the ARM920T as an ongoing example. Chapter 6 presents and discusses the results of the experiments conducted using the Mälardalen benchmark programs. Chapter 7 goes through related work in the area of WCET analysis and compares it to the METAMOC method. Chapter 8 and 9 conclude the thesis and present possibilities for future work, respectively.

Chapter 2

Prerequisites

This chapter introduces the various prerequisites that are needed for understanding the rest of the thesis: RTSs, WCET analysis, UPPAAL-specific model checking and weighted push-down systems (WPDSs). The section about hard RTSs is based on Chapter 1 and 13 in [13], while the section about WCET and methods for finding WCETs is based on [44]. The third section is about model checking using UPPAAL and is based on [10]. The chapter's final section provides a theoretical, and to some extent practical, introduction to the formalism of WPDSs, which we use for doing static analysis on programs, specifically value analysis.

2.1 Real-Time Systems and Scheduling

RTSs are systems that must perform an action within a specified time interval. Examples of such systems are airbags, missile systems and advanced hospital equipment. A RTS consist of software processes that are executed concurrently to produce an output. Each process has, among other properties, a priority and a deadline. The priority of a process is a measure of how important the process is compared to the other processes in the system. The deadline specifies the amount of time units that are allowed to elapse from the process is started ("released") until it must have completed. If the process has not completed when the deadline is reached, it is said to have missed its deadline. RTSs where processes are allowed to miss deadlines are called soft RTSs, whereas RTSs that do not allow deadlines to be missed are called hard RTSs. Only hard RTSs are considered in this thesis.

The planning of process execution is called scheduling. In the domain of scheduling, processes are either periodic or sporadic. A periodic process is released repeatedly with a fixed time interval. A sporadic process can be released at any time, e.g. they can be triggered by external events. The most widely used scheduling approach, Fixed-Priority Scheduling (FPS), works by assigning static priorities to processes pre-runtime, and on runtime the process with the highest priority is running, while the other processes are waiting. When allowing both periodic and sporadic processes, however, a process with a low priority might be allowed to continue execution, even though a high priority sporadic process has been released. This is due to the fact that FPS is a non-preemptive scheduling scheme, which do not stop ("preempt") the running process, i.e. the running process is always allowed to finish.

For FPS, a simple, optimal priority assignment scheme called rate monotonic priority assignment exists. This scheme needs the WCET of all processes in the system that are about to be scheduled. The task of finding WCETs of processes is done by a WCET analysis.

2.2 Worst-Case Execution Time Analysis

We start by introducing the concept WCET and subsequently introduce two classes of methods for finding WCETs.

2.2.1 Worst-Case Execution Time

The WCET of a process is the longest execution time of all possible execution times of the process. This is illustrated in Figure 2.1. The figure shows an example of distributions of execution times of a process. All execution times in the "SAFE" area are safe, i.e. not underapproximated, WCETs. In general the approximated WCETs should be as close to the real WCET as possible, i.e. as close to the bound as possible. WCETs that satisfy this property are called sharp WCETs. If the WCETs are not sharp, the utilisation of the system will not be high. Generally, a high utilisation is desired, as this might allow the use of a cheaper processor for a particular system or facilitate the execution of bigger systems on existing processors.

To be able to find the WCET of a process, the process must terminate at some point, and the input to the process must be bounded, i.e. infinite input is not allowed. In addition, it is often required that the programmer provides loop bounds and other forms of annotations.

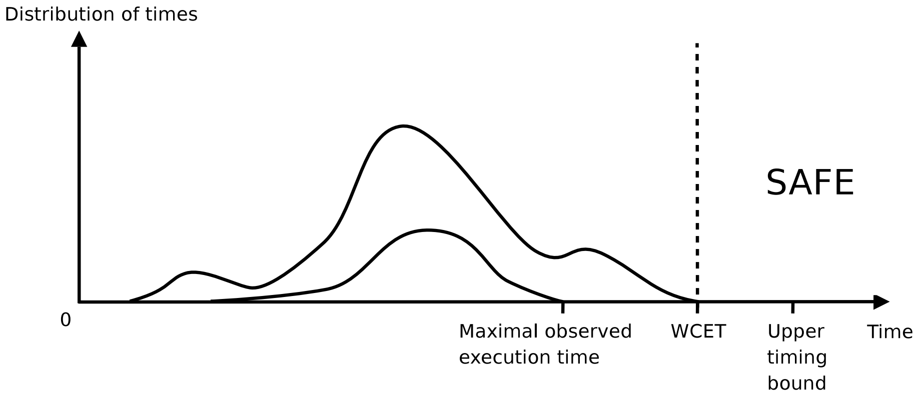


Figure 2.1: Distribution of execution times of a process.

2.2.2 Classes of WCET Analysis Methods

In this section we discuss two classes of methods for finding the WCET of a process. One way to approximate the WCET of a process is to run the process a number of times and observe the execution time and select the maximal observed execution time. This will usually not be the true WCET, as the true WCET might not be observed. Even though this approach is unsafe, it is the principle of measurement-based methods for finding WCETs. Contrary to measurement-based methods, static methods are able to determine safe upper bounds for the WCET.

Measurement-Based Methods

Measurement-based methods find the WCET by executing the process and measuring the time required for the process to complete. The advantage of this approach is that it is relatively simple to implement — the code just needs to be run and its execution time measured. Furthermore, the approach does not need any considerable adaption to be applied to new hardware architectures.

The main disadvantage is that the methods based on the approach typically only find a maximal observed execution time, which can be less than the actual WCET, as can be seen in Figure 2.1 on the preceding page. Another disadvantage is that the process needs to be executed several times, which can be time-consuming if the process itself is time-consuming. To find more precise estimates, more advanced methods have been developed. One method is to measure the time of all basic blocks of a process. Basic blocks are sequential blocks of code that do not contain branching, i.e. they contain no if-statements or loops. The WCET of each basic block together with the time required for branches is summed up to find the WCET of the entire process.

Static Methods

Static methods do not execute the process but is instead based on analysis of the process' code and some model of the system on which the process is planned to be executed. The methods are typically used to find an upper bound on the execution time of a process, which is strictly greater than the WCET of the process. They are often based on data flow analysis, which is a well-known technique in the domain of compilers. The class is divided in two groups: one using static analysis and one using model checking. The METAMOC method presented in this thesis combines static analysis and model checking and therefore lies in the intersection of the two groups.

2.3 Model Checking using UPPAAL

UPPAAL is a model checker for real-time systems [10]. It has been developed since 1995 by researchers at Aalborg University, Denmark, and Uppsala University, Sweden. In this section we introduce the features in UPPAAL that are relevant to the modelling done in Chapter 5 starting on page 51. The primary source for this section is [10].

To model interactions in the real world accurately, UPPAAL uses network of timed automatas (NTAs). The automata interact by synchronising with each other through synchronisation channels. Compared to ordinary timed automata defined in e.g. [8], the timed automata in UPPAAL have only one initial state and are extended with the following features:

- **Automata templates**, which make it possible to have several instances of the same automaton defined with different parameters.
- **Constants**, which are declared as `const name value`. Constants must have integer values.
- **Bounded integer variables**, which are declared as `int[min,max] name`. If `[min,max]` is omitted, the default range is $[-32768, 32767]$. Expressions involving integers may be used in guards (constraints on constants, integers and clocks), invariants and assignments. The bounds are checked on verification time, where states with bound violations are discarded.
- **Binary synchronisation channels**, which are declared as `chan name`. An edge labelled with `name!` synchronises with an edge labelled with `name?`. If several synchronisation pairs are possible, UPPAAL chooses a pair non-deterministically.
- **Broadcast channels**, which are declared as `broadcast chan name`. This enables a single sender, which has `name!` on an edge, to synchronise with an arbitrary number of receivers, which have `name?` on an edge. Broadcast sending does not block the sender, as the sender can take the `name!` action even though there are no receivers.
- **Urgent synchronisation channels**, which are declared by prefixing a channel declaration with `urgent`. If a synchronisation transition on an urgent channel is enabled, delays must not occur. No clock guards are allowed on edges that use urgent synchronisation channels.
- **Urgent locations**, which are semantically equivalent to locations with an added clock `x` and an added invariant `x<=0`, where `x` is set to zero on all incoming edges. Consequently, delays are not allowed in an urgent location.
- **Committed locations**, which are even stricter than urgent locations. A state in the state space of a NTA is committed if any of the locations in the state is committed. A committed state cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.
- **Arrays** of clocks, channels, constants and integers. Arrays are declared by appending a size to the variable name, e.g. `clock a[2];`, `chan c[3];`, or `int[0,100] i[5];`.
- **Initialisers**, which makes it possible to initialise integer variables and arrays of integers, e.g. `int i := 10;`, or `int ia[3] := {0, 42, 60};`.

Expressions in UPPAAL can involve clocks, constants and integer variables. Four types of expression labels exist: guards, synchronisations, assignments and invariants. Guards, synchronisations and assignments are used on edges,

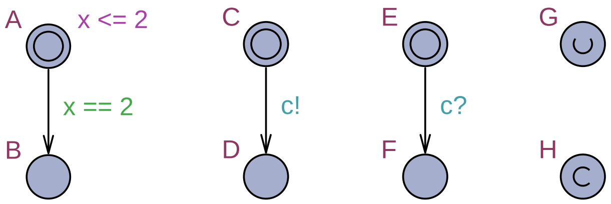


Figure 2.2: Some of the automata elements in UPPAAL.

whereas invariants are used in locations. The precise behaviour of a NTA in UPPAAL is defined by the semantics in [10, Definition 3].

Figure 2.2 shows some of the automata elements in UPPAAL. Locations `A` and `B` form a timed automaton, where `A` is the initial location, `B` is the terminal location, `A` has an outgoing edge to `B`, `A` has the invariant $x \leq 2$, and the edge connecting `A` and `B` has the guard $x = 2$. The invariant and the guard both involve the clock `x`, and the semantics is that the automaton must stay in location `A` for two time units and then move to `B`. Locations `C` and `D` form a timed automaton, in which the edge is able to synchronise through a channel `c` with the edge in the automaton formed by locations `E` and `F`. Location `G` is an urgent location, and location `H` is a committed location.

In UPPAAL, the properties that make up system specifications are called queries. UPPAAL's query language is a derived version of timed computation tree logic (TCTL). Compared to TCTL, or at least the version of TCTL presented in [8], the query language in UPPAAL includes the modal operators $\Box$ (globally, **G**) and $\Diamond$ (finally, **F**) but does not allow nesting of path formulae. We end this chapter with three examples of queries in UPPAAL, where the last example is used by the METAMOC method:

- `E<> Train1.Crossing and Train2.Crossing`: A path exists in which we eventually end up in a state, where the automata `Train1` and `Train2` are in the location `Crossing` at the same time. Without further information about the system, this is probably not a desirable property to satisfy.

- `A[] not deadlock`: The system never deadlocks ("for all paths it holds on the entire subsequent path that the system does not deadlock").

- `sup: cyclecounter`: This is a recent addition to the query language found in the development version of UPPAAL. To help us avoid running UPPAAL's verification engine several times to obtain a WCET using our method, the UPPAAL developer Alexandre David added the query type during March 2009. The query makes UPPAAL output the maximum possible value for the clock `cyclecounter`.

2.4 Weighted Push-Down Systems

WPDSs are a way to perform inter-procedural, control-flow sensitive, and possibly data-sensitive, program analysis. In this project we use WPDSs for our value analysis, estimating the value contained in registers at a point of execution. We have chosen to use the framework of WPDSs, because it is a quite general framework, and it is possible to create a new analysis with relatively modest effort. It is at the same time very powerful and easily extendable, e.g. with more context for the analysis. It is also very good for representing procedural imperative programs, because a program's call stack can be precisely represented. Queries can even be qualified with regards to the stack.

The theory on WPDSs presented here is based on [36]. A WPDS is a generalisation of a push-down system (PDS), with "weights" added to each edge. Weights are a general black-box abstraction for program data, which conform to the mathematical model of a semi-ring, and can thus record many important properties about a program's behaviour. In the following, we give a definition of a PDS, a definition of a regular set of configurations in a PDS, a result regarding computing *pre** and *post**, a definition of a bounded idempotent semi-ring (a weight domain), and finally a definition of a WPDS. We will then present a classical example of a WPDS instantiation.

Definition 1 (Push-Down System)**.** *A PDS is a triple $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of control states, Γ is a finite set of stack symbols and $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ is a finite set of rules detailing how transitions in the system can happen.*

A configuration is a control state combined with a string of stack symbols, e.g. $\langle p, u \rangle$, where $p \in P$ and $u \in \Gamma^*$.

A rule $r \in \Delta$ is written as $\langle p, \gamma \rangle \hookrightarrow \langle p', u \rangle$, where $p, p' \in P, \gamma \in \Gamma$ and $u \in \Gamma^*$.

The rules define a transition relation $\Rightarrow$ on configurations of $\mathcal{P}$ as follows: If $r = \langle p, \gamma \rangle \hookrightarrow \langle p', u' \rangle$ then $\langle p, \gamma u \rangle \Rightarrow \langle p', u'u \rangle$ for all $u \in \Gamma^*$.

Given a PDS $\mathcal{P}$ and a set of configurations C, we define:

- The set of configurations that can lead to a configuration in C:
  pre$^*(C) = \{c' | \exists c \in C : c' \Rightarrow^* c\}$.
- The set of configurations that can follow from a configuration in C:
  post$^*(C) = \{c' | \exists c \in C : c \Rightarrow^* c'\}$.

where $\Rightarrow^*$ is the reflexive and transitive closure of $\Rightarrow$.

In the framework of WPDSs, one restricts the sets of configurations considered to only the regular sets of configuration. Regular sets of configurations (even infinite) can be described in a finite manner using $\mathcal{P}$-automata. This is done using finite automata describing configurations of a PDS $\mathcal{P}$.

Definition 2 (Regular Sets of Configurations)**.** *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$, we define a $\mathcal{P}$-automaton to be a 5-tuple $(Q, \Gamma, \rightarrow, P, F)$, where $Q \supseteq P$ is a finite set of states, Γ is the finite alphabet (the stack symbols from the PDS $\mathcal{P}$), $\rightarrow \subseteq Q \times \Gamma \times Q$ is a transition relation, P is the set of initial states (the states from the PDS $\mathcal{P}$) and $F \subseteq Q$ is the set of final states.*

A configuration of $\mathcal{P}$, $\langle p, u \rangle$, is said to be accepted by a $\mathcal{P}$-automaton, if the automaton can accept the word $u = u_1 \dots u_n$ when started in the state p (that is $p \xrightarrow{u_1} \cdots \xrightarrow{u_n} q$, with some $q \in F$).

A set of configurations, C, is said to be regular if it is accepted by a $\mathcal{P}$-automaton.

In the following we will assume, without loss of generality, that all rules in a PDS push at most two symbols on the stack. We note that rules pushing more than two symbols can be rewritten into a number of rules using additional states.

Having now defined a model for regular sets of configurations, we now show how *pre** and *post** can be computed for such sets, when they are represented as $\mathcal{P}$-automata.

Theorem 1 (Decidability and Complexity of Deciding *pre** and *post**)**.** *Given a regular set of configurations C, both pre$^*(C)$ and post$^*(C)$ can be computed in polynomial time.*

Proof outline. This proof outline is based closely on [36, p. 9]. Given a PDS $\mathcal{P}$ and a $\mathcal{P}$-automaton $\mathcal{A}$ recognising the set C, $\mathcal{P}$-automata $\mathcal{A}_{pre^*}$ and $\mathcal{A}_{post^*}$ can be constructed, such that $\mathcal{A}_{pre^*}$ recognises the set of configurations $pre^*(C)$ and $\mathcal{A}_{post^*}$ recognises the set of configurations $post^*(C)$.

$\mathcal{A}_{pre^*}$ is constructed as follows:

If $\langle p, \gamma\rangle \hookrightarrow \langle p', w\rangle$ in $\mathcal{P}$, where $w = w_1 \dots w_n$, and $p' \xrightarrow{w_1} \cdots \xrightarrow{w_n} q$ in the current automaton, then add a transition $p \xrightarrow{\gamma} q$ to the current automaton. Repeat until no more transitions can be added.

$\mathcal{A}_{post^*}$ is constructed as follows:

1. For each rule of the form $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ add a new state $p'_{\gamma'}$.

2. Add transitions using the following three rules, until saturation:

   - If $\langle p, \gamma\rangle \hookrightarrow \langle p', \epsilon\rangle$ in $\mathcal{P}$ and $p(\xrightarrow{\epsilon})^* \xrightarrow{\gamma} (\xrightarrow{\epsilon})^* q$ in the current automaton, then add a transition $p' \xrightarrow{\epsilon} q$.
   - If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\rangle$ in $\mathcal{P}$ and $p(\xrightarrow{\gamma'})^* \xrightarrow{\gamma} (\xrightarrow{\epsilon})^* q$ in the current automaton, then add a transition $p' \xrightarrow{\gamma'} q$.
   - If $\langle p, \gamma\rangle \hookrightarrow \langle p', \gamma'\gamma''\rangle$ in $\mathcal{P}$ and $p(\xrightarrow{\gamma'})^* \xrightarrow{\gamma} (\xrightarrow{\epsilon})^* q$ in the current automaton, then add the two transitions $p' \xrightarrow{\gamma'} p'_{\gamma'}$ and $p'_{\gamma'} \xrightarrow{\gamma''} q$.

For details that the described algorithms are correct and of polynomial time complexity, see [39].

□

Before we can define a WPDS we will need to define which kinds of weights can be associated with the edges, and how these weights can interact. This is defined using the general framework of a bounded idempotent semi-ring, which has many useful instantiations. We now give the formal definition of a bounded idempotent semi-ring and show an instantiation, to exemplify the many new concepts.

Definition 3 (Bounded Idempotent Semi-Ring (Weight Domain))**.** *A bounded idempotent semi-ring*[1] *is a 5-tuple: $(D, \oplus, \otimes, \bar{0}, \bar{1})$, where*

D *is a set of weights (the elements in the semi-ring),*

$\oplus$ *is the combine operation for the weight domain ("combine" two elements of the semi-ring),*

[1] A semi-ring is a ring without the requirement of inverse elements.

$\otimes$ *is the extend operation for the weight domain ("extend" an element with another element), and*

$\bar{0}$ *and* $\bar{1}$ *are the zero- and one-element, respectively.*

In addition, the following must hold:

1. $(D, \oplus)$ *must be a commutative monoid*[2] *with* $\bar{0}$ *as its neutral element, and where* $\oplus$ *is idempotent. In particular this implies the following properties:*

   Closure: $\forall a, b \in D : a \oplus b \in D.$

   Associativity: $\forall a, b, c \in D : (a \oplus b) \oplus c = a \oplus (b \oplus c).$

   Commutativity: $\forall a, b \in D : a \oplus b = b \oplus a.$

   Identity element: $\forall a \in D : \bar{0} \oplus a = a \oplus \bar{0} = a.$

   Idempotency: $\forall a \in D : a \oplus a = a.$

2. $(D, \otimes)$ *must be a monoid with* $\bar{1}$ *as its neutral element. This implies the following properties:*

   Closure: $\forall a, b \in D : a \otimes b \in D.$

   Associativity: $\forall a, b, c \in D : (a \otimes b) \otimes c = a \otimes (b \otimes c).$

   Identity element: $\forall a \in D : \bar{1} \otimes a = a \otimes \bar{1} = a.$

3. $\otimes$ *distributes over* $\oplus$*:* $\forall a, b, c \in D : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ *and* $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c).$

4. $\bar{0}$ *is an annihilator with respect to* $\otimes$*, i.e.* $\forall a \in D : a \otimes \bar{0} = \bar{0} = \bar{0} \otimes a.$

5. *In the partial order* $\sqsubseteq$ *defined as* $a \sqsubseteq b \iff a \oplus b = a$*, there are no infinite descending chains.*

An example of a weight domain is the weight domain **MINPATH** $= (\mathbb{N}_0 \cup \{\infty\}, min, +, \infty, 0)$. In **MINPATH** the elements are the natural numbers $\{0, 1, 2, \ldots\}$ and ∞. The combine operation is the min-operation, that is defined by $min(a, b) = \begin{cases} a & \text{if } a < b \vee b = \infty \\ b & \text{otherwise} \end{cases}$.
The extend operation is simply addition on the natural numbers (also allowing ∞, such that $\forall a \in \mathbb{N}_0 : a + \infty = \infty$).

We will now define a WPDS and show how the **MINPATH** weight domain can be used to find the minimal path length from one WPDS configuration to another.

Definition 4 (Weighted Push-Down System)**.** *A WPDS is a triple* $\mathcal{W} = (\mathcal{P}, \mathcal{S}, f)$*, where* $\mathcal{P} = (P, \Gamma, \Delta)$ *is a PDS,* $\mathcal{S} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ *is a weight domain and* $f : \Delta \to D$ *is a mapping function, mapping the PDS's transitions to elements in the weight domain.*

Given a sequence of rules $\sigma \in \Delta^*$ *(which can be seen as a path in the associated push-down automata), we associate a value from the weight domain using the extend operation as follows: If* $\sigma = [r_1, \ldots r_k]$*, the associated weight is* $v(\sigma) = f(r_1) \otimes \ldots \otimes f(r_k)$*.*

[2] A monoid is a group without the requirement of inverse elements.

For any two configurations c and c' we let $path(c, c')$ denote the set of all rule sequences that transforms c into c'.

For regular sets of configurations S and T, we define the ***meet-over-all-valid-paths*** *value as* $MOVP(S, T) = \bigoplus\{v(\sigma) | \exists s \in S, t \in T : \sigma \in path(s, t)\}$

An example program and a corresponding WPDS are provided in Figure 2.3. The WPDS demonstrates how the stack can be used to accurately model the inter-procedural control flow.

```
 1  main () {
 2    x = 5;
 3    y = 7;
 4    x = f(x, y);
 5    if (x > 5) {
 6      y = 8;
 7    }
 8  }
 9
10  f (a, b) {
11    if (a + b < 10) {
12      return 7;
13    } else {
14      return 9;
15    }
16  }
```

$$\langle p, n_{main}\rangle \hookrightarrow \langle p, n_2\rangle$$
$$\langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle$$
$$\langle p, n_3\rangle \hookrightarrow \langle p, n_4\rangle$$
$$\langle p, n_4\rangle \hookrightarrow \langle p, n_f n_5\rangle$$
$$\langle p, n_5\rangle \hookrightarrow \langle p, n_6\rangle \qquad \langle p, n_5\rangle \hookrightarrow \langle p, n_7\rangle$$
$$\langle p, n_6\rangle \hookrightarrow \langle p, n_7\rangle$$
$$\langle p, n_7\rangle \hookrightarrow \langle p, \epsilon\rangle$$

$$\langle p, n_f\rangle \hookrightarrow \langle p, n_{11}\rangle$$
$$\langle p, n_{11}\rangle \hookrightarrow \langle p, n_{12}\rangle \quad \langle p, n_{11}\rangle \hookrightarrow \langle p, n_{14}\rangle$$
$$\langle p, n_{12}\rangle \hookrightarrow \langle p, \epsilon\rangle$$

$$\langle p, n_{14}\rangle \hookrightarrow \langle p, \epsilon\rangle$$

Figure 2.3: An example program, in a C-like syntax, and a WPDS accurately modelling the (data-insensitive) control flow.

Using the **MINPATH** weight domain we can find the minimal path length, from one configuration to another. If we assign the weight 1 (not to be confused with the one-element $\bar{1}$) to every rule in the WPDS, we can use $MOVP(S, T)$ to determine the shortest path length from one set of configurations to another. For example:

$$\begin{gathered}
MOVP(\{\langle p, n_{main}\rangle\}, \{\langle p, \epsilon\rangle\}) = \\
\bigoplus\{v([\langle p, n_{main}\rangle \hookrightarrow \langle p, n_2\rangle, \langle p, n_2\rangle \hookrightarrow \langle p, n_3\rangle, \langle p, n_3\rangle \hookrightarrow \langle p, n_4\rangle, \\
\langle p, n_4\rangle \hookrightarrow \langle p, n_f n_5\rangle, \langle p, n_f\rangle \hookrightarrow \langle p, n_{11}\rangle, \langle p, n_{11}\rangle \hookrightarrow \langle p, n_{12}\rangle, \\
\langle p, n_{12}\rangle \hookrightarrow \langle p, n_\epsilon\rangle, \langle p, n_5\rangle \hookrightarrow \langle p, n_7\rangle, \langle p, n_7\rangle \hookrightarrow \langle p, n_\epsilon\rangle,]), v(\ldots), \cdots\} = \\
\bigoplus\{1 \otimes 1 \otimes 1 \otimes 1 \otimes 1 \otimes 1 \otimes 1 \otimes 1 \otimes 1, \cdots\} = \\
\bigoplus\{9, \cdots\} = \\
min(9, min(\cdots)) = 9.
\end{gathered}$$

Of course, in an implementation it would be intractable to calculate directly on the definitions as done here, due to the enumeration of all possible paths from one regular set to another. Indeed efficient algorithms exist for computing the *MOVP* over regular sets, in much the same way as pre^* and $post^*$ works for PDSs [36, sec. 3.3].

Chapter 3

Optimisation Techniques in Modern Processors

Modern processors use a number of optimisation techniques to speed up average performance. Many of these techniques will, however, make the worst-case performance much harder to predict, because they introduce an element of non-determinism or a very high dependence on the previous state (which might be unknown or hard to predict).

In this chapter some of these techniques will be described. The descriptions are based on [42, 34]. Other sources are referenced when used.

This chapter is organised into four sections. The first two sections cover caching and pipelining, which are advantageous to consider when finding the WCET of a process on a specific platform [13, p. 647]. The third section covers timing anomalies, a phenomenon that might appear with some of the optimisation techniques or combinations thereof. The final section covers abstract representations of caches and pipelines.

3.1 Caching

Caching is an optimisation technique used to bridge the large speed difference between the processor and the main memory. The speed up is achieved by placing a smaller but much faster memory chip, called a cache, between the main memory and the processor. Caching takes advantage of the principle of locality. More explicitly there are two types of locality: temporal locality, stating that if a data item has just been used it will probably be used again soon after, and spatial locality, stating that if a data item has just been used data items nearby will probably be needed soon after. The cache holds a copy of recently used parts from main memory. If a part of memory is not in the cache and needs to be fetched from main memory into the cache, it is called a cache miss. The opposite is called a cache hit.

A cache has a number of important parameters, affecting its performance and predictability:

The capacity of the cache is the number of bytes that can be stored in the cache. The capacity is of course very important, since it limits the amount of main memory that can be cached, and thus given quick access to. The smaller the cache, the slower the execution, in general.

The cache line size is the number of bytes that are transfered from or to main memory in one transfer. The cache consist of $k = \frac{capacity}{cache\ line\ size}$ cache lines. Main memory M is split into memory blocks $m_1, m_2, \ldots, m_n \in M$ for the purpose of caching. Given a memory block m_i the address of the memory block can be found by the function $adr(m_i) = i - 1$. For instance, the address of $adr(m_1) = 0$. For simplicity, we assume that the size of memory blocks and cache lines is the same.

The associativity of the cache determines in which cache lines a memory block can reside. "Fully associative" means that a memory block can reside in any cache line. The opposite, "direct mapped", means that a memory block can reside in precisely one cache line. A trade-off is A-way set associative caches, which partition the cache into k/A disjunct sets, called cache sets, and maps each memory block to exactly one set. Cache sets is denoted $s_1, s_2, \ldots, s_{k/A}$. The n'th cache line in a cache set is called "way n". Common A-way set associative caches are for $A \in \{2, 4, 8\}$. The associativity is a trade-off between quick lookup (direct mapped being the fastest) and effectiveness of the cache in achieving more hits. Fully associative and direct mapped are special cases of an A-way set associative cache with $A = k$ or $A = 1$, respectively.

An example of a 2-way set associative cache can be seen in Figure 3.1. The cache lines, in which a memory block can reside, can be calculated by $cacheSet(m) = \{l_i \mid i = \left(adr(m) \bmod \frac{k}{A}\right) \cdot A + j\}$, where m is a memory block and $j \in \{1, \ldots, A\}$.

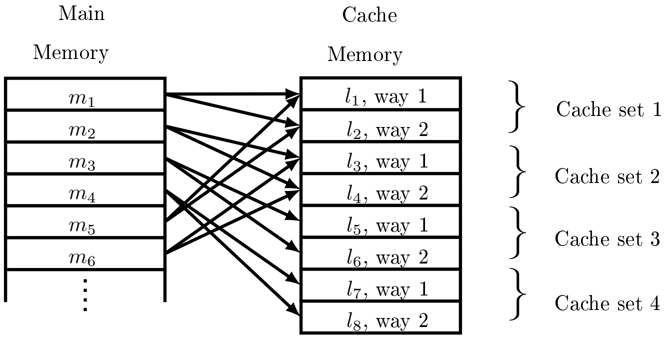


Figure 3.1: Illustration of which memory blocks can reside in which cache lines in a 2-way set associative cache. The arrows represent which cache lines a memory block can be cached in.

The replacement policy determines which memory block to evict, when a new memory block needs to enter the cache, and the cache is already full. The policy has a very large impact on the predictability of the cache. Some common policies are: first-in first-out (FIFO), least-recently used (LRU), pseudo round-robin (PRR) and pseudo least-recently used (PLRU).

	Write Back	**Write Through**
Write Allocate	Write to cache now, write to memory later	Write to cache now, write to memory now
No Write Allocate	N/A	Only write to memory, update cache if needed

Table 3.1: Summary of the different ways of handling writes. Write allocate/write back and no write allocate/write through are the most commonly used policies.

Handling writes is an important aspect of designing a cache, which can have a huge impact on the overall performance [34, p. 483]. The problem can be divided into two cases: what to do on write hits and what to do on write misses. Write hits occur when a data item in a memory block has to be modified and the memory block is in the cache. Write misses happen when the memory block is not in the cache. For write hits the simplest approach is to modify the memory block in the cache and write the change directly to main memory. This approach is called write through. The drawback is that this affects performance as every write will cause a write to main memory. Another more complicated approach is to only modify the memory block in the cache and only write the memory block to main memory when it is evicted from the cache. This means we now need to keep track of which memory blocks have to be written back to memory when they are evicted from the cache. This is usually done by using a "dirty" bit for each cache line to indicate whether the memory block has been modified or not. For write misses, the simplest approach is to only write the change to main memory. This method is called "no write allocate". A more complex approach is "write allocate", where the memory block is fetched into the cache on writes. By the temporal locality principle, the memory block will probably soon be used again. Often hardware using write back uses write allocate and hardware using write through uses no write allocate to avoid complicating the hardware design. The different options are summarised in Table 3.1.

The notation of caches and cache replacement policies have been inspired by [6, 20, 11, 19]. There are generally two types of cache designs: a unified cache or separate instruction and data caches. In a unified cache, the space used for data versus the space used for instructions is balanced without additional work. Separate instruction and data caches can provide faster access times since they can be accessed simultaneously. Furthermore, it is possible to adjust the parameters of the caches for different needs. For instance, it is uncommon to modify instructions and therefore they need not be written back to main memory. Due to the different advantages, it is common to have both separate instruction and data caches *and* a unified cache, where the latter stores both instructions and data memory blocks. This is done by having multiple levels of caches. For instance, processors often have separate level 1(L1) caches and then have one or more unified larger and slower L2/L3/... caches. If more than one level of caches are used, another property becomes important: are the caches exclusive or inclusive? An inclusive cache means that e.g. the L2 cache contains everything the L1 cache contains, plus some more. An exclusive cache means that the content of the caches are totally disjunct.

3.1.1 Cache Replacement Policies

Cache replacement policies are used for choosing which memory block in the cache to evict. Making an optimal decision for this can be hard as this requires knowledge of which memory blocks will be used in the future execution. Based on the temporal locality principle, an ideal policy keeps recently used memory blocks in the cache and evicts the least recently used memory block. This is exactly what the LRU replacement policy does. A nice property of this policy is that unused memory blocks will eventually be removed from the cache. This property also holds for the FIFO policy. It does, however, not hold for replacements policies as PRR and PLRU, which makes these harder to predict.

An example demonstrating the LRU and FIFO replacement policies will be given along with formal definitions. In the definitions of the replacement policies, a cache is considered a set $L = \{l_1, l_2, \ldots, l_k\}$ of cache lines, and $M = \{m_1, m_2, \ldots, m_n\}$ is the set of all memory blocks. We extend M with the empty element I, which represents a cache line that does not contain any memory block: $M' = M \cup \{I\}$. The element is also called an invalid cache line. The capacity k and associativity A is usually a power of two and will in the following be assumed to be so. Initially, a concrete cache state is defined:

Definition 5 (Concrete Cache State)**.** *A concrete cache state is a mapping $c : L \rightarrow M'$, and C_c is the set of all concrete cache states.*

For each of the replacement policies a concrete update function is defined which describes the effect of a memory access to the memory block m with the concrete cache state c.

The order of the cache lines is used to model the age of memory blocks in the concrete cache state. The least recently used memory block is placed in cache line l_1. The policies are described for a fully associative cache. The interested reader can find a formal definition of an A-way set associative LRU policy in [6].

LRU is often considered the ideal policy, and other replacement policies are usually compared to LRU. The LRU replacement policy can be divided into two cases: cache hits and cache misses. In case of a cache hit the memory block accessed should be marked as the most recently used memory block. In the second case the memory block that is accessed is not in the cache and the least recently used memory block should be evicted. An example of the first case that demonstrates the concrete LRU update function can be seen in Figure 3.2 on the next page.

The figure shows a concrete cache state c, which is updated to a new concrete cache state c' when the memory block m_2 is accessed. It should be noted that m_2 is the second oldest memory block in c. As m_2 is accessed again, it should be the youngest memory block in c'. Another consequence is that the memory blocks that are younger than m_2 in c should have their age increased by one in c'. The age of memory blocks that are older than m_2 in c keep their age in c'.

The second case can be seen in Figure 3.3 on the facing page. The figure shows how a concrete cache state c is updated when the memory block

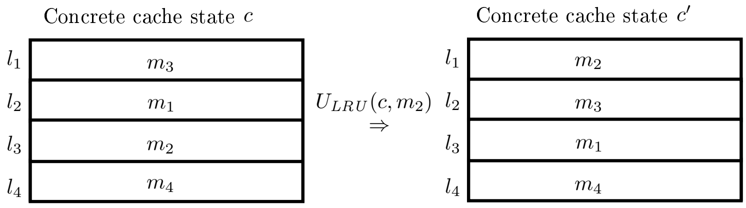


Figure 3.2: Update of concrete cache state c to c' in the case of a cache hit. The LRU replacement policy is used to update the ages of memory blocks in the cache.

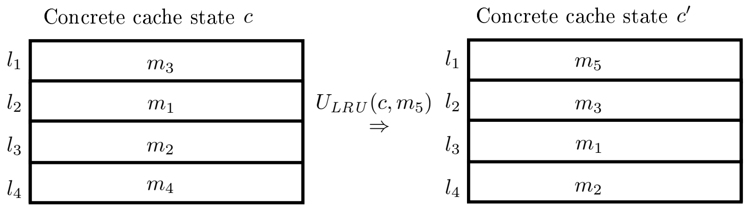


Figure 3.3: Update of concrete cache state c to c' in the case of a cache miss. The LRU replacement policy is used to choose which memory block to evict.

m_5, that is not in c, is accessed: a cache miss. In c, m_3 is the youngest memory block and m_4 is the oldest. Since m_5 is not already in the cache, and m_4 is the oldest, m_4 will be evicted from the cache and m_5 will be the youngest memory block in c'.

Definition 6 (Concrete LRU Cache Update Function)**.** *The update function $U_{LRU} : C_c \times M \to C_c$ takes a concrete cache state and a memory block as input and produces a concrete cache state as output. $U_{LRU}(c, m) = c'$, where c' is defined as:*

$$c' = \begin{cases} [l_1 \mapsto m, & \\ \quad l_i \mapsto c(l_{i-1}) \mid i \in \{2, \dots, h\}, & \\ \quad l_i \mapsto c(l_i) \mid i \in \{h+1, \dots, A\}] & \textit{if } \exists l_h : c(l_h) = m \\ [l_1 \mapsto m, & \\ \quad l_i \mapsto c(l_{i-1}) \mid i \in \{2, \dots, A\}] & \textit{otherwise.} \end{cases}$$

FIFO is a simpler replacement policy than LRU. Therefore, the main advantage of FIFO is that the update logic is cheaper to implement. The update logic can be implemented with a round-robin counter for each cache set that points to the oldest memory block. That said, FIFO is quite similar to LRU. It can also be divided into two similar cases: one for cache hits and one for cache misses. An example of the first case, where a memory block in the cache is accessed, can be seen in Figure 3.4 on the next page. Since FIFO only uses a counter to represent the oldest memory block, c and c' are equal on cache hits.

The second case for cache misses of a memory block m is equivalent to

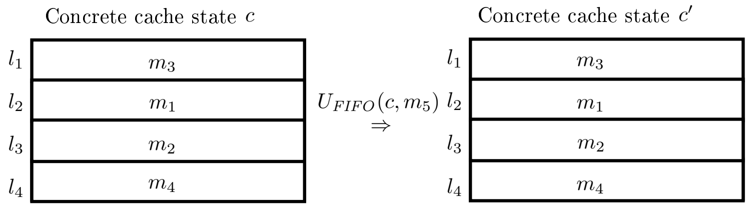


Figure 3.4: Update of concrete cache state c to c' in the case of a cache hit. The FIFO replacement policy is used to update the ages of memory blocks in the cache, i.e. do nothing in this case.

that of LRU. The oldest memory block will be evicted from the cache, and m will become the youngest memory block.

Definition 7 (Concrete FIFO Cache Update Function). *The update function $U_{FIFO} : C_c \times M \to C_c$ takes a concrete cache state and a memory block as input and produces a concrete cache state as output. $U_{FIFO}(c, m) = c'$, where c' is defined as:*

$$c' = \begin{cases} c & \textit{if } \exists l_h : c(l_h) = m \\ [l_1 \mapsto m, & \\ \;\; l_i \mapsto c(l_{i-1}) \mid i \in \{2, \ldots, A\}] & \textit{otherwise.} \end{cases}$$

3.2 Pipelining

This section is based on [42, 34] and we assume basic knowledge of processor architectures.

Pipelining is a well-known concept for instance in the car industry, where cars are assembled on a pipeline with specialised workers or robots carrying out the same action repeatedly. To understand how this can be applied to a simple processor, it should be realised that executing a single instruction can easily take more than one cycle. For instance, executing an instruction with two operands, on the simple processor modelled in Figure 3.5 on the facing page, is done by setting the instruction type on the ALU operation line and waiting for the operands to be made available in Data register1 and Data register2. When the operands are available, the instruction is executed. After the instruction has been executed, and as soon as any previous result in the write register has been written to cache or main memory, the result is stored in the write register. Assuming that fetching an instruction from the cache takes one cycle, fetching an operand from the cache takes one cycle, executing the instruction takes one or two cycles depending on the instruction type, and writing the result takes one cycle. In total this is five to six cycles for executing a single instruction.

Applying the concept of pipelining to processors means that the tasks of the processor are divided into stages which are then carried out consecutively. An example this division into stages could be: “Instruction Fetch”, “Instruction Decode”, “Operand Fetch”, “Instruction Execution”, and “Write Back”. All stages are performed in parallel and are designed with the intention to work on an instruction during each cycle. In Figure 3.6 on the next page such a five stage

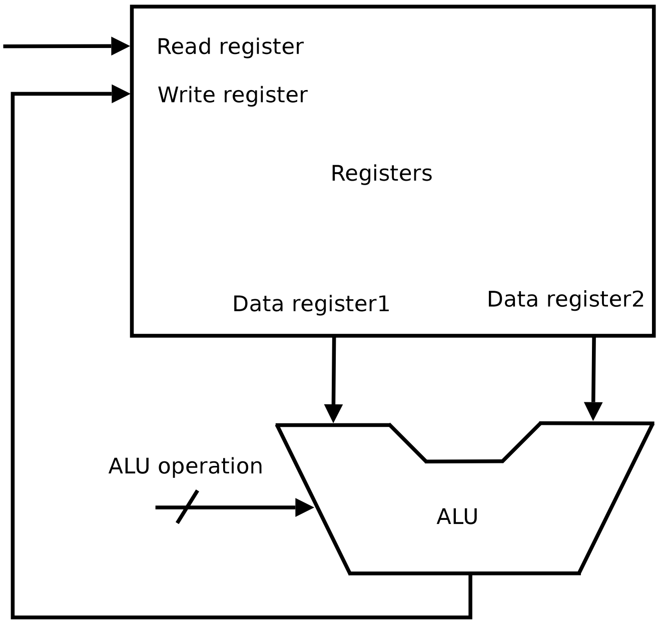


Figure 3.5: Simplified processor model

pipeline is shown. More complex pipelines might have more than one unit in a stage, e.g. the "Instruction Execution" stage might consist of an integer unit and a floating point unit.

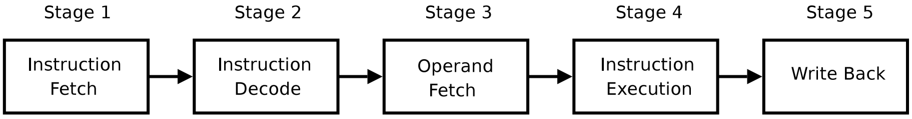


Figure 3.6: Five stage pipeline.

The advantage of using pipelining is an overall increase of the number of instructions executed per time unit. The time taken by an instruction to flow through a non-pipelined data path limits the cycle frequency of the processor, while the shorter data paths in the stages allow the cycle frequency to be increased dramatically. Conceptually, if a processor's data path is not pipelined, each instruction must flow through all the "stages" before allowing the next instruction to enter. Using a five stage pipeline, ideally an instruction can be completed each cycle. This can be seen in Table 3.2 on the following page.

At this point it might seem tempting to increase the number of stages in a pipeline, however, this will usually not yield a much better performance. The reason is that in practice instruction prefetching might not work as planned and instead introduce pipeline stalls. A pipeline stall is a situation where the pipeline cannot perform useful work but instead performs `NOPs` (No OPerations).

The code in Figure 3.7 on the next page introduces a pipeline stall. Table 3.3 shows what happens in the pipeline for each cycle. In the two first cycles

Cycle	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5
1	Instr. 1				
2	Instr. 2	Instr. 1			
3	Instr. 3	Instr. 2	Instr. 1		
4	Instr. 4	Instr. 3	Instr. 2	Instr. 1	
5	Instr. 5	Instr. 4	Instr. 3	Instr. 2	Instr. 1
6	Instr. 6	Instr. 5	Instr. 4	Instr. 3	Instr. 2
⋮	⋮	⋮	⋮	⋮	⋮

Table 3.2: The first six cycles of a five stage pipeline.

```
while (i != 42) {
        if (j == 1) {
                j = 42;
        }
        i++;
}
```

(a)

```
1   While:  CMP i, 42
2           BNE Next
3           NOP
4           CMP j, 1
5           BNE Ncon
6           NOP
7   Con:    MOV j, 42
8   Ncon:   NOP
9           ADD i, i, 1
10          BR While
11          NOP
12  Next:   ...
```

(b)

Figure 3.7: Prefetching example.

everything is fine, but in cycle three a NOP is fetched in the Instruction Fetch stage. The reason to have NOPs in the assembly code is to keep the program correct. The problem is that the pipeline fetches the NOP before it decodes the BNE (Branch if Not Equal) instruction, and then realises that it might actually need to fetch the instruction at line 12 of part (b) of Figure 3.7 instead. Rather than always performing NOPs after branches, a better idea is to perform useful work. This technique is called a delay slot. A delay slot is an instruction that is placed after branches, and the instruction is executed regardless of the branch. It is then the task of the compiler to try to place a useful instruction in the delay slot, or a NOP in case no useful instruction can be placed there. In case of conditional branches such as BNE, delay slots are not the only problem, since the pipeline has to stall and wait for the result of executing the CMP instruction in the Instruction Execution stage. Before CMP can be executed, the operands (in this case i) need to be fetched from the cache or main memory. In Table 3.3 on the facing page i is assumed to be in the cache. Even though there is a cache hit, the Instruction Fetch stage does not know the result of executing the CMP instruction in cycle four, which means it stalls and has to insert a NOP instruction. If the pipeline had even more stages this would result in stalling for even more cycles. One technique to decrease the number of pipeline stalls is branch prediction.

Cycle	Instr. Fetch	Instr. Decode	Operand Fetch	Instr. Execution	Write Back
1	`CMP i, 42`				
2	`BNE Next`	`CMP i, 42`			
3	`NOP`	`BNE Next`	`CMP i, 42`		
4	`NOP`	`NOP`	`BNE Next`	`CMP i, 42`	
...	...				

Table 3.3: Example of pipeline stall as a result of branching.

3.2.1 Branch Prediction

Branch prediction is an optimisation technique which can be used to reduce the amount of pipeline stalls. We introduce it through a small example. In Figure 3.8 a code snippet and the corresponding assembly code can be seen. If this code is executed on a pipelined processor, and the variable `i` is not cached, the pipeline has to stall until `i` is fetched from memory, since the processor does not know whether to prefetch the instruction in either line three or five in part (b).

```
1  ...
2  if (i == 0) {
3      j = 42;
4  } else {
5      k = 42;
6  }
7  ...
```

(a)

```
1          CMP i, 0
2          BNE Ncon
3  Con:    MOV j, 42
4          BR Next
5  Ncon:   MOV k, 42
6  Next:   ...
```

(b)

Figure 3.8: Code snippet and corresponding assembly code.

Another way of handling branches in a pipelined processor is to let the fetch stage continue fetching, rather than stalling, and flushing the preceding stages, when it is realised that the branch is to be taken. This might be costly if the branch is taken, because the flushing of stages means that the pipeline will be idle for several cycles.

If the value of `i` in line one could somehow be predicted, it would be possible to prefetch the instructions of line three or five. This is, however, not possible in general. Instead of giving up, this has given rise to a number of different branch prediction techniques.

One idea is to statically predict that branches are never taken. This only causes problems when branches are indeed taken, and in this case the computation would have to be "undone". Two common rule exist. Firstly, if the target of a conditional branch is backward, then the branch is taken. The reason is that this indicates a loop and loops are generally iterated several times. Secondly, if the target of a conditional branch is forward the branch is not taken. The reason for this is that forward branches are sometimes used in error checking and errors rarely happen. This prediction is not as good as the one for backward branches, as many forward conditional branches are not related to error checking.

Address	**Prediction bit(s)**
0xDEAD	*taken*
0xBEEF	*not taken*
...	...

Table 3.4: Example of a history table.

Since conditional branches are often inside loops, a noticeable performance decrease can result if branches are repeatedly mispredicted. To avoid this, dynamic branch prediction has been designed to learn from past experiences. A common technique for learning is to use a history table as seen in Table 3.4.

The history table is used similarly to a cache. Some history tables use a single bit to represent if the branch was taken last time, or not. This leads to a common, unfortunate behaviour with nested loops, as the last iteration over the inner loop will flip the prediction bit. This means that the next time the inner loop is entered, the loop will most likely be mispredicted. Consequently, branch prediction can give very good results but entails some challenges.

3.2.2 Out-of-Order Execution

Another technique to reduce the number of pipeline stalls is to execute instructions out-of-order. In effect, the constraint that instructions should be be executed in sequential order is no longer valid, although the sequential semantics must be maintained. For instance, in the code snippet in Figure 3.9, line one might lead to a pipeline stall if the memory block containing `i` is not in the cache. If the memory block containing `j` is in the cache, it would be more efficient to execute line two before line one.

```
1  DIV R5, i, 3
2  MUL R3, j, 2
3  ADD R1, R3, R2
4  ADD R2, R4, 3
5  ADD R1, R2, 4
```

Figure 3.9: Assembly code snippet demonstrating out-of-order execution where Rx is a register.

Out-of-order execution requires the pipeline to be redesigned. An overview of an idea for a new design can be seen in Figure 3.10 on the next page, where the Instruction Fetch and Instruction Decode stages have been collapsed. Another change is that the Instruction Execution stage usually has two or more units that perform parallel out-of-order execution. The collapsed stage works in-order and makes a "pool" of instructions available to the units in the Instruction Execution stage. After instructions have been executed, the results are committed back to the cache or main memory. This is done in the commit stage, which is in-order. The Operand Fetch stage has been renamed to Reservation Station. The Reservation Station is responsible for making operands available and resolving conflicts.

Executing instructions out-of-order and in parallel gives rise to three types of

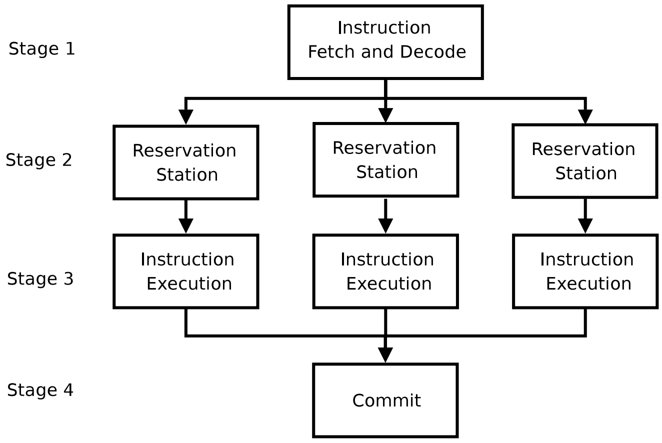


Figure 3.10: Overview of a processor design with out-of-order execution.

data dependencies, resulting in conflicts on an out-of-order execution processor. The data dependencies are demonstrated in the code in Figure 3.9 on the facing page:

- Read after write (RAW) can be seen in line two and three, where R3 will be written to in line two and the result is read in line three. The RAW dependency blocks out-of-order execution of line three before line two.
- Write after read (WAR) can be seen in line three and four, where R2 needs to be read in line three, before it can be written in line four.
- Write after write (WAW) can be seen in line three and five, as both line three and five write their result to register R1, but the result of line five must be the last to be written to R1.

The reader might have noticed that both WAR and WAW could be overcome by having additional, so-called "secret" registers to store the result in. For instance, the WAR example could be solved by first copying the value of R2 to another register, that would be used by the instruction in line three. This would allow both instructions to be executed out of order. This technique is generally known as register renaming and is performed by the Instruction Fetch and Decode stage, Reservation Stations, and the Commit stage. The technique is able to eliminate many WAR and WAW dependencies [42, p. 280].

3.2.3 Speculative Execution

Branch prediction and out-of-order execution are not sufficient for avoiding all pipeline stalls. For instance, out-of-order execution does not work well if there are many branches and the basic blocks are small. Branch prediction does not always help enough, if, for instance, a memory block needs to be fetched from main memory or a long floating point operation has to complete before the result of a conditional branch is known.

Speculative execution can improve the average performance of execution. This is done by executing instructions without knowing if they are actually

Label	Dispatch cycle	Instruction
A	1	`LDR R4, [R3]`
B	2	`ADD R5, R4, R4`
C	3	`ADD R11, R10, R10`
D	4	`MUL R12, R11, R11`
E	5	`MUL R13, R12, R12`

Table 3.5: Instructions for timing anomaly example [31].

going to be executed. This might, however, also result in slowdowns. For instance, if speculation results in execution of instructions that require memory blocks that are not in the cache, thereby causing a cache miss, as the memory blocks have to be fetched from main memory. To prevent this scenario, some modern processors have special speculative instructions which only use cached memory blocks.

Another problem with speculative execution is that executing instructions without strictly obeying checks in conditional branches might cause overflows or exceptions that would not otherwise happen. This is an undesirable situation, and it needs to be resolved in hardware. A way to do this is to add a "poison" bit to each register, set the bit when a register stores speculative results and only raise exceptions when it is confirmed that an exception truly will happen.

3.3 Timing Anomalies

A processor exhibits timing anomalies when the local execution time of a single instruction has a counter-intuitive influence on the global execution time of a program [16]. For instance, a cache hit rather than a cache miss at a particular point of execution might yield a *longer*, rather than a shorter, overall program execution time. Timing anomalies are of crucial relevance to WCET analysis, as it is in general more simple and efficient to assume that local worst-case decisions produce the global WCET.

We introduce timing anomalies through a concrete example [31], where a cache hit triggers an overall longer execution time than a cache miss. The execution takes place on a simplified processor model with three parallel resources: a load-store unit (LSU), an integer unit (IU) and a multi-cycle integer unit (MCIU). Table 3.5 contains an instruction sequence that will be executed on the model. The second column shows in which cycles the instructions are dispatched. The `LDR` (load to register) instruction uses the LSU, `ADD` uses the IU, and `MUL` uses the MCIU. The IU permits out-of-order execution, whereas the LSU and the MCIU do not. For each instruction, the leftmost register is the destination register, while the other registers are source registers.

The instructions' use of registers makes them dependent. For example, the `LDR` instruction must have loaded a value into register `R4`, before the subsequent `ADD` is able to use the register as a source for addition. Using labels, it is clear that B depends on A, D depends on C, and E depends on D. All the dependencies are RAW dependencies as explained in Section 3.2.2 on page 32. The `LDR` instruction executes for two cycles if there is a cache hit and ten cycles otherwise. The `ADD` and `MUL` instructions execute for one and four cycles,

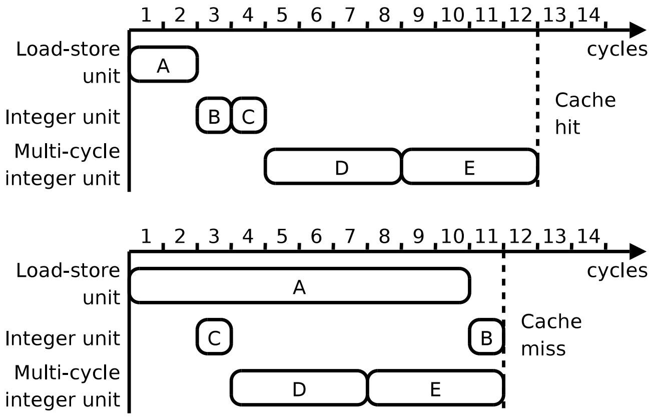


Figure 3.11: Timing anomaly example [16, 31].

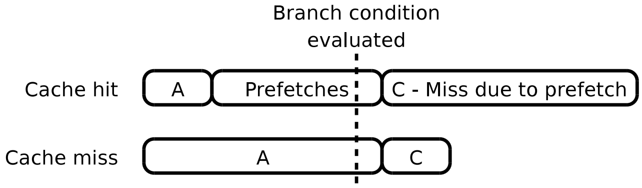


Figure 3.12: Speculation anomaly example [35].

respectively.

Figure 3.11 shows the execution of the instruction sequence, with and without a cache hit for the `LDR` instruction. Consider the upper half of the figure. Instruction B is dispatched at cycle two, but due to its dependency on A, execution of B has to wait until cycle three, where A is done executing. At cycle three, C is dispatched, but the processor's policy causes B to be executed first, as it has been waiting and therefore is older. In the figure's lower half, instruction A experiences a cache miss and delays the execution of B until cycle 11. As the IU permits out-of-order execution, C is executed in cycle three, immediately after having been dispatched. The instruction sequence exposes a timing anomaly for the processor model, as the execution with a cache hit has a longer overall execution time than the execution with a cache miss.

Consequently, on this processor a WCET analysis that assumes local worst-case behaviour, i.e. cache miss, for each instruction possibly underapproximates the WCET.

The example illustrated in Table 3.5 on the facing page and Figure 3.11 is called a scheduling anomaly. Another example is a speculation anomaly, which occurs with branch prediction [35]. A mispredicted branch can cause unnecessary instruction fetching, polluting the cache. If the first instruction in the mispredicted branch is a cache miss, then the branch condition might be evaluated before the mispredicted branch can cause more harmful fetches. This anomaly is illustrated in Figure 3.12.

A third type of timing anomalies, also mentioned in [35], is cache timing anomalies. Contrary to the other two types, these are anomalies caused entirely by strange cache behaviour, i.e. there is no out-of-order execution, branch prediction or other optimisation techniques involved. It is notable that some in-order architectures, such as the Motorola ColdFire 5307, do exhibit speculation and cache timing anomalies, thus to avoid anomalies it is not enough to avoid out-of-order execution [35, 43].

Timing anomalies had only been informally defined until 2006, where Becker et al. put forward a definition in [35]. The definition is inspired by a series of observations with regards to the underlying hardware model, the desired degree of abstraction and the necessary code level to inspect. Because the definition takes all these elements into account, it has the advantage that it covers all types of timing anomalies, even possibly future types. The obvious drawback is that it is not a single definition, it is a framework of complex definitions and is therefore not simple to apply. Also, the definitions have not yet, to our knowledge, been the basis for any published studies on timing anomalies.

Since timing anomalies makes it unsafe to find the WCET of an instruction sequence by assuming local worst-case behaviour for each instruction, WCET analyses relying solely on that approach are not able to cope with timing anomalies. In [31], Lundquist and Stenström present two methods for dealing with timing anomalies, called "the pessimistic serial-execution method" and "the program modification method". The former method assumes that all instructions are executed one at a time and sum up all individual instruction WCETs together with a cache miss penalty for each instruction. Because main memory is substantially slower than cache, all these extra cache miss penalties lead to a very overapproximated WCET. The latter method modifies the instruction sequence in a hardware dependent way, such that its execution on the particular hardware does not exhibit timing anomalies. The modification includes the insertion of special instructions to force the processor to do in-order execution in problematic parts of the instruction sequence.

3.4 Abstract Representations

Contrary to representing the behaviour of caches and pipelines concretely, abstract representations are an efficient, yet relatively precise, way of representing sets of concrete states. In this section we present the research on cache and pipeline abstraction that we have studied.

3.4.1 Abstraction of Caches

In Section 3.1 on page 23, we presented a way of representing caches in a concrete manner. In this section we will describe how to represent sets of concrete caches in a space-efficient, abstract manner. This is useful for analyses that try to predict cache behaviour at different execution points in a program. The analyses work on the control flow graph (CFG) of a process. Each node in the CFG is assigned an abstract cache state, which is a representation of the various concrete cache states that the processor might be in at that execution point. In this section we present a definition of an abstract cache state, along with may and must analyses for the LRU and FIFO cache replacement policies. The definition of LRU and examples are based on [6, 19, 20]. In this section we assume fully associative caches, but note that the definitions could easily be extended to A-way set-associative caches.

Definition 8 (Abstract Cache State)**.** *An abstract cache state is a function $\hat{c} : L \rightarrow 2^M$, where $L = \{l_1, \ldots, l_k\}$ is the set of cache lines, and M is the set of memory blocks. The mapping indicates the maximal age of an item in the cache. $\hat{C}_c$ denotes the set of all abstract cache states. We use k to denote the number of cache lines, $|L|$.*

For example, assuming two cache lines, l_1 and l_2, and two memory blocks, m_1 and m_2, the abstract cache state $\{l_1 \rightarrow \emptyset, l_2 \rightarrow \{m_1, m_2\}\}$ is a representation of the concrete cache states $\{l_1 \rightarrow m_1, l_2 \rightarrow m_2\}$ and $\{l_1 \rightarrow m_2, l_2 \rightarrow m_1\}$.

When a CFG node has more than one incoming transition, a join function is used to determine a sound approximation of the cache contents at this CFG node.

Definition 9 (Join Function)**.** *A join function is a mapping from two abstract cache states to single abstract cache state:*

$$JOIN : \hat{C}_c \times \hat{C}_c \rightarrow \hat{C}_c.$$

As mentioned before, a cache analysis is usually divided into two separate analyses: a must analysis, used to predict which memory blocks are definitely in the cache, and a may analysis, used to predict which memory blocks are definitely not in the cache. Join functions will be defined for both analyses.

Must Analysis

The must analysis is fundamental to cache analysis, as it provides information on which memory blocks are in the cache at a given execution point. This improves the results of a WCET analysis by predicting definite cache hits.

Before presenting abstract update functions, we present a join function which is used for both LRU and FIFO replacement policies.

Definition 10 (LRU/FIFO Must Join Function)**.** *For LRU and FIFO replacement polices the join function is defined as*

$$JOIN(\hat{c}_1, \hat{c}_2) = \{l_i \rightarrow X | y \in X \Leftrightarrow \exists j, k : y \in \hat{c}_1(l_j) \wedge y \in \hat{c}_2(l_k) \wedge i = max(j, k)\}$$

I.e. the age that a memory block in the cache can have at a node is the maximal age that the item has in its predecessors.

In the following we present abstract update functions for the replacement policies.

LRU's abstract must update function is almost similar to the concrete update function for LRU. The difference is that cache lines can contain sets of memory blocks. It must therefore be determined how cache hits affect other memory blocks in the same cache line containing the memory block being accessed — for the must analysis the ages are upper bounds, meaning that it is safe to let the other memory blocks stay. In Figure 3.13

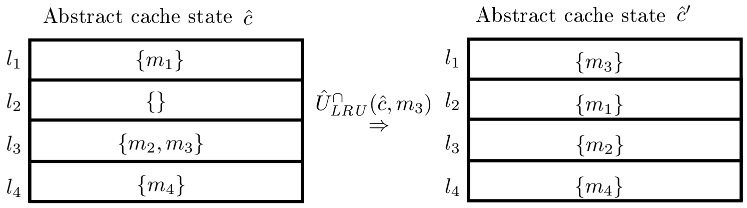


Figure 3.13: An example of the abstract must update function for LRU in use.

an example demonstrating the abstract must update function for LRU is shown.

A formal definition of the abstract must LRU cache update function is given below.

Definition 11 (Abstract Must LRU Cache Update Function)**.** *The update function, $\hat{U}^{\cap}_{LRU} : \hat{C}_c \times M \rightarrow \hat{C}_c$, takes an abstract cache state and a memory block as input and produces an abstract cache state as output. $\hat{U}^{\cap}_{LRU}(\hat{c}, m) = \hat{c}'$, where $\hat{c}'$ is defined as:*

$$\hat{c}' = \begin{cases} [l_1 \mapsto \{m\} & \\ \;\; l_i \mapsto \hat{c}(l_{i-1}) | i \in \{2, \ldots, h-1\}, & \\ \;\; l_h \mapsto \hat{c}(l_{h-1}) \cup (\hat{c}(l_h) \setminus \{m\}), & \\ \;\; l_i \mapsto \hat{c}(l_i) | i \in \{h+1, \ldots, A\}] & \textit{if } \exists l_h : m \in \hat{c}(l_h) \\ [l_1 \mapsto \{m\}, & \\ \;\; l_i \mapsto \hat{c}(l_{i-1}) | i \in \{2, \ldots, A\}] & \textit{otherwise.} \end{cases}$$

The update function has two cases: cache hit and cache miss. On a cache miss the memory block accessed is moved to the first cache line, and the remaining cache lines are moved down one line (with the last cache line dropping out). On a cache hit, the memory block accessed is moved to the first cache line, and the cache lines from the first to the one where the hit was are moved one down.

FIFO's update logic is a bit more simple than that of LRU and the abstract update function is therefore even more similar to the concrete update function. The difference between the concrete and the abstract update function is that the latter uses abstract cache states. A definition of the abstract must FIFO cache update function is given below.

Definition 12 (Abstract Must FIFO Cache Update Function)**.** *The update function $\hat{U}^{\cap}_{FIFO} : \hat{C}_c \times M \rightarrow \hat{C}_c$ takes an abstract cache state and a memory block as input and produces an abstract cache state as output. $\hat{U}^{\cap}_{FIFO}(\hat{c}, m) = \hat{c}'$ where $\hat{c}'$ is defined as:*

$$\hat{c}' = \begin{cases} \hat{c} & \textit{if } \exists l_h : m \in \hat{c}(l_h) \\ [l_1 \mapsto \{m\}, & \\ \;\; l_i \mapsto \hat{c}(l_{i-1}) | i \in \{2, \ldots, A\}] & \textit{otherwise.} \end{cases}$$

Again the update function has two cases: cache hit and cache miss. On a cache hit the cache is not altered. On a cache miss the memory block is moved to the first cache line, and the remaining cache lines are moved one down, with the last falling out.

May Analysis

The may analysis is used to find all memory blocks that could possibly be in the cache at a given point. This can be used to predict if a memory block is definitely not in the cache.

For the may analysis the following join function is used:

Definition 13 (LRU/FIFO May Join Function)**.** *For the LRU and FIFO replacement policies the may join function is defined as*

$$JOIN(\hat{c}_1, \hat{c}_2) = \{l_i \rightarrow X | y \in X \Leftrightarrow \exists j, k : y \in \hat{c}_1(l_j) \wedge y \in \hat{c}_2(l_k) \wedge i = min(j, k)\}$$

I.e. the age that a memory block in the cache can have at a node is the minimal age the item has in its predecessors.

The replacement policies are presented below.

LRU's abstract may update function is different from the abstract LRU must update function. In Figure 3.14 an example is presented which demonstrates the abstract may update function.

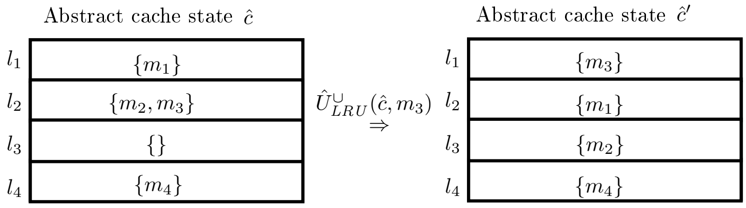


Figure 3.14: An example of the abstract may update function for LRU in use.

A formal definition of the abstract may LRU cache update function is given below.

Definition 14 (Abstract May LRU Cache Update Function)**.** *The update function $\hat{U}^{\cup}_{LRU} : \hat{C}_c \times M \rightarrow \hat{C}_c$ takes an abstract cache state and a memory block as input and produces an abstract cache state as output. $\hat{U}^{\cup}_{LRU}(\hat{c}, m) = \hat{c}'$ where $\hat{c}'$ is defined as:*

$$\hat{c}' = \begin{cases} [l_1 \mapsto \{m\}, & \\ \quad l_i \mapsto \hat{c}(l_{i-1}) | i \in \{2, \ldots, h\}, & \\ \quad l_{h+1} \mapsto \hat{c}(l_{h+1}) \cup \big(\hat{c}(l_h) \setminus \{m\}\big), & \\ \quad l_i \mapsto \hat{c}(l_i) | i \in \{h+2, \ldots, A\}] & \textit{if } \exists l_h : m \in \hat{c}(l_h) \\ [l_1 \mapsto \{m\}, & \\ \quad l_i \mapsto \hat{c}(l_{i-1}) | i \in \{2, \ldots, A\}] & \textit{otherwise.} \end{cases}$$

FIFO's abstract may cache update function is the same as for the abstract must cache update function. In case of cache hits both the may and the must FIFO update functions do not change the abstract cache state. In case of cache misses, the accessed memory block is inserted as the first and the age of all other memory blocks are increased by one. A formal definition of $\hat{U}^{\cup}_{FIFO}$ is therefore omitted.

3.4.2 Abstraction of Pipelines

Due to the complexity of the stages in a pipeline and the interdependencies between the stages, it is difficult to make a useful pipeline abstraction. For example, in [38] Ferdinand et al. define an abstract pipeline state simply as a set of concrete pipeline states.

Chapter 4

Hardware Platform

The WCET of a process depends highly on the hardware platform it is executed on. The hardware platform is also known as the execution environment, although this in some instances also covers more than just the hardware, e.g. the operating system as well [13, p. 635].

As mentioned in Chapter 3, there are two primary optimisation techniques of modern processors, which are advantageous to consider when finding the WCET of a process on a specific hardware platform: caching and pipelining. In practice there might be more elements to consider, e.g. it might be necessary to take the misalignment of busses into account. Misalignment occurs, since the speed of the main memory bus often is slower than the bus between the processor and caches. This can be safely approximated by using a larger constant on each memory access, however, to get sharper WCETs this must in some cases be modelled more precisely [23].

Due to the dependence of WCET analysis on a hardware platform, we have chosen a specific processor as a basis for the implementation of METAMOC. We have chosen a processor from the well-known ARM9 family from ARM Ltd.: the ARM920T 32 bit processor. The ARM920T has been chosen due to its widespread use in embedded systems. For example, it powers the Sun SPOT Wireless Sensor Network node, the FIC Neo FreeRunner mobile phone, and is used in many custom embedded systems. Even though we have chosen to implement the method for a specific hardware platform, the method's applicability is not limited. For instance, the method is designed to be easily extendable to other cache types, or other pipelines.

For the ARM families the relation between System-on-Chip, processor and processor core is illustrated in Figure 4.1. The figure shows that a System-on-Chip, or microcontroller, contains among other things a processor and some amount of RAM. The relevant parts of a processor, in the context of this report, are the processor core and caches.

The processor core in the ARM920T processor is the ARM9TDMI. This core is used in all processors in the ARM9 family, which, besides the ARM920T, includes the ARM922T and ARM940T processors [25, 40]. It features a five-stage pipeline, divided in the stages: Fetch, Decode, Execute, Memory, and Writeback [40]. The pipeline is illustrated in Figure 4.2.

The processor has separate 16 KB instruction and 16 KB data caches [28]. Each cache is 64-way set associative and has 8 words (32 bytes) per cache line.

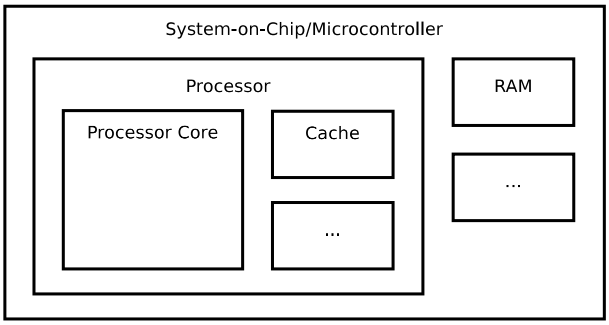


Figure 4.1: The relation between System-on-Chip, processor and processor core.

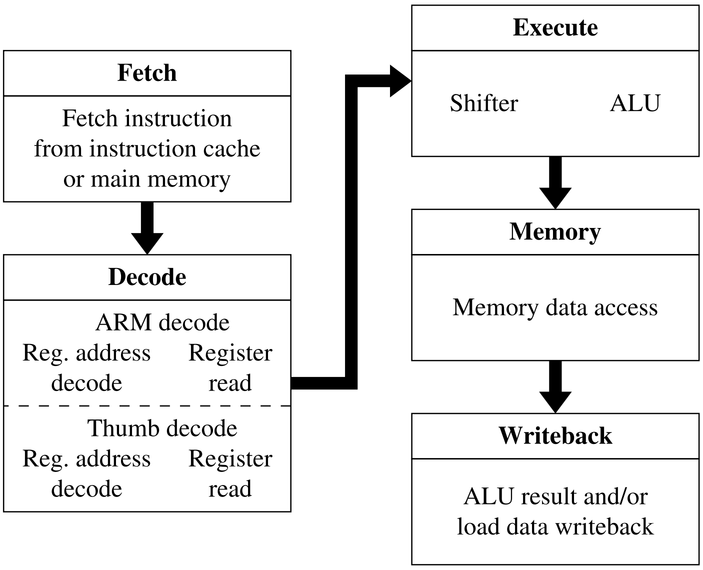


Figure 4.2: The five-stage pipeline in the ARM9TDMI processor core [40].

Both caches feature a choice between FIFO or pseudo random as replacement policy.

Both 32 bit ARM instructions and 16 bit Thumb instructions are handled by the ARM9TDMI processor core [27]. The Thumb instruction set improves the code density at the cost of being more limited than ARM instructions with regard to possible instruction operands. The decode stage is divided into two parts: one for ARM and one for Thumb. Only one of them is active at a time. We currently only handle ARM instructions, but support for the Thumb instruction set can easily be added. The following section gives a brief introduction to the ARM assembly language.

4.1 ARM Assembly Language

WCET analysis must be done on the machine code level of a process, since this is the only level with enough information [20]. At the machine code level, compiler optimisations have been performed and the memory addresses for static data is specified directly. Machine code can be represented either in binary form or as assembly language, where the latter is typically chosen such that the code can be inspected by humans. Since the machine code level is unavoidable when doing WCET analysis, it is necessary to be acquainted with the ARM assembly language when creating an analysis method for ARM processors.

The language has instructions for branching, data processing, status register transferring, loading, storing, co-processing and exception generation [29, 22]. In addition, the language supports conditional execution of all instructions. Table 4.1 shows the subset of ARM instruction mnemonics we have encountered while compiling the Mälerdalen WCET benchmark programs [4] to the ARM920T processor using GCC.

We will now explain in some detail the most common ARM assembly language constructs we have encountered: function calls, function returns, conditional instructions, data operations and memory operations.

Function calls are usually carried out using the BL instruction, which saves the address of the next instruction to register LR and then jumps to the static address given as argument:

```
BL 832C
```

The called functions usually returns using the BX instruction, which jumps to the address stored in the register given as argument:

```
BX LR
```

In functions which use register LR, its value is typically stored on the stack and later restored by POP {LR} just before BX LR.

Conditional instructions are preceded by instructions, such as CMN, CMP and TST, which set condition flags for the instructions to react on. Arithmetic, logical, shifting and rotating instructions can be suffixed with an "S", e.g. ASRS, EORS or MULS, and will then also set the condition flags. For instance, the following two instructions compare the value in register R0 to the decimal value 100 and jump to the static, hexadecimal address 8428 if the compared values are equal:

Mne.	Meaning
ADC	Add with Carry
ADD	Add
AND	Logical AND
ASR	Arithmetic Shift Right
ASRS	Arithmetic Shift Right and set condition flags
B$\langle cc \rangle$	Branch on Condition
B	Unconditional Branch
BIC	Bit Clear
BL$\langle cc \rangle$	Conditional Branch with Link
BL	Unconditional Branch with Link
BX	Unconditional Branch and Exchange
CMN	Compare (Negative)
CMP	Compare
EOR	Exclusive OR
EORS	Exclusive OR and set condition flags
LDM	Load Multiple
LDR	Load Register (Word)
LDRB	Load Register (Byte)
LDRH	Load Register (Halfword)
LDRSH	Load Register (Halfword) and set condition flags
LSL	Logical Shift Left
LSLS	Logical Shift Left and set condition flags
LSR	Logical Shift Right
LSRS	Logical Shift Right and set condition flags
MLA	Multiply Accumulate
MLS	Multiply and Substract
MOV	Move
MUL	Multiply
MULS	Multiply and set condition flags
MVN	Logical NOT
ORR	Logical OR
ORRS	Logical OR and set condition flags
POP	Pop from the stack
PUSH	Push on the stack
RSB	Reverse Subtract
RSBS	Reverse Subtract and set condition flags
SMLAL	Mult. Accum. Signed Long
SMULL	Multiply Signed Long
STM	Store Multiple
STR	Store Register (Word)
STRB	Store Register (Byte)
STRH	Store Register (Halfword)
STRSH	Store Register (Halfword) and set condition flags
SUB	Subtract
TST	Test
UMLAL	Mult. Accum. Unsigned Long
UMULL	Multiply Unsigned Long

Table 4.1: The ARM instruction mnemonics relevant to our work [29, 22].

```
CMP R0, #100
BEQ 8428
```

Branching to dynamic addresses can be carried out by either using the BX instruction with a register as argument (like when returning from functions) or by writing to the program counter PC.

The ADD and AND instructions serve well as general examples of the arithmetic and bitwise instructions, respectively. The first of the following two instructions stores the sum of the values in R0 and R1 in R2, whereas the next instruction performs a bitwise AND of the values in R3 and R4 and saves the result in R5:

```
ADD R2, R0, R1
AND R5, R3, R4
```

The MOV instruction copies values between registers. For instance, the following two instructions load R0 with the decimal value 100 and copy the value of R1 into R2:

```
MOV R0, #100
MOV R2, R1
```

Loading and storing of registers can be performed for bytes, halfwords (two bytes), words (four bytes) and multiple words (a multiple of four bytes). The following instruction loads R0 with the contents of memory pointed to by the address stored in R1:

```
LDR R0, [R1]
```

The POP and PUSH instructions are aliases for load multiple and store multiple, respectively. The POP <reglist> instruction is the canonical form of LDM SP!, <reglist>, whereas PUSH <reglist> is the canonical form of STMDB SP!, <reglist> [29]. The <reglist> argument is a list of registers separated by commas and enclosed in curly brackets. The STMDB instruction is a STM postfixed by DB, meaning that the stack pointer SP is decremented before the store is carried out. The LDM and STM instructions can also be postfixed by IA (increase after), IB (increase before) and DA (decrease after). The default is IA, which does not need to be specified.

The following instruction is a variation of the previous load instruction, where the value in R1 is used as base address and the value in R2 is used as offset, i.e. the memory address is the sum of values in the two registers:

```
LDR R0, [R1, R2]
```

It is also possible to bitwise shift or rotate the value of a register before using it. The ARM instruction set offers logical shift left (LSL), logical shift right (LSR), arithmetic shift right (ASR), rotate right (ROR) and rotate right extended (RRX) [22]. Contrary to the logical shifts, ASR maintains the sign of the value. The RRX modifier takes no argument and rotates a value through the Carry flag (one of the processor's status flags) by one bit, i.e. the Carry flag is moved into the most significant bit, and the least significant bit is moved into the Carry flag. Compared to the previous LDR instruction, the following instruction shifts the value of R2 left by two bits before using it to calculate the memory address:

Mne.	Meaning	Mne.	Meaning
`AL`	Always (normally omitted)	`LE`	Signed less than or equal
`CC/LO`	Carry Clear/Unsigned lower	`LS`	Unsigned lower or same
`CS/HS`	Carry Set/Unsigned higher or same	`LT`	Signed less than
		`MI`	Negative
`EQ`	Equal	`NE`	Not equal
`GE`	Signed greater than or equal	`PL`	Positive or zero
`GT`	Signed greater than	`VC`	No overflow
`HI`	Unsigned higher	`VS`	Overflow

Table 4.2: Mnemonics for conditional execution of ARM instructions [29].

```
LDR R0, [R1, R2, LSL #2]
```

Despite the complexity of the expression, only the value of `R0` is changed.

Table 4.2 shows the mnemonics for conditional execution of instructions. The mnemonics can be used with `B` and `BL`, as explicitly indicated in Table 4.1, but they can actually be used with all instructions by suffixing the condition mnemonic. For instance, the conditional subtraction (`SUBGT`) in the following instruction sequence is not executed, since 5 is not greater than 10, and thus the value of `R2` remains the decimal value 20:

```
MOV R0, #5
MOV R1, #10
MOV R2, #20
CMP R0, R1
SUBGT R2, R0, R1
```

The instructions in the ARM instruction set takes a different number of cycles to execute and are handled in different stages of the pipeline [27, Chapter 7]. In other words, all instructions flow through the entire pipeline, and the instruction type decides in which stage most cycles are spent. Even if a stage is not relevant to an instruction, it always spends at least one cycle in the stage before moving on. Data operations, such as addition and bitwise AND, are executed in the execute stage, where they remain for a number of cycles depending on whether `PC` is the destination and whether there is any shifting or rotation involved. Multiplication is a special data operation, where the number of spent cycles in the execute stage depends on the operands. Load and store instructions are executed in the memory stage and their required number of cycles depend on the amount of data to load and the varying access time to cache or main memory. Table 4.3 shows the instruction cycle times for the instructions relevant to our work.

From Table 4.3 it is clear that the cycle time for the multiplication instructions depend on the value of m. This value is determined by the multiplier operand (MO), which is the source register marked by `Rs` in the Multiply instructions list in [29, page 2]. For example, for `MUL` the MO is the rightmost source register, but this is not a general pattern. For `MLA`, `MUL`, `SMLAL` and `SMULL`, the value of m is

- 1 if bits [31:8] of the MO are all zero or one,

Instruction	Cycles	Comment
Data oper.	1	Normal case, PC not destination
Data oper.	2	With register controlled shift, PC not destination
Data oper.	3	PC destination register
Data oper.	4	With register controlled shift, PC destination
LDR	1	Normal case, not loading PC
LDR	2	Not loading PC and following instruction uses loaded word (1 cycle stall)
LDR	3	Loaded byte, half-word, or unaligned word used by following instruction (2 cycle stall)
LDR	5	PC is destination register
STR	1	All cases
LDM	2	Loading one register, not PC
LDM	n	Loading n registers, $n > 1$, not including PC
LDM	$n + 4$	Loading n registers, $n > 0$, including PC
STM	2	Storing one register
STM	n	Storing n registers, $n > 1$
B, BL, BLX	3	All cases
MLA, MUL	$2 + m$	All cases
SMLAL, SMULL, UMLAL, UMULL	$3 + m$	All cases

Table 4.3: The number of cycles required to execute some of the most commonly used instructions in the ARM instruction set on the ARM9TDMI processor core. The data operations (including the different types of multiplication) are executed in the execute stage of the core's pipeline, whereas the load and store operations are executed in the memory stage. [27, Section 7.1]

- 2 if bits [31:16] of the MO are all zero or one,
- 3 if bits [31:24] of the MO are all zero or all one, or
- 4 otherwise.

For UMLAL and UMULL, the value of m is

- 1 if bits [31:8] of the MO are all zero,
- 2 if bits [31:16] of the MO are all zero,
- 3 if bits [31:24] of the MO are all zero, or
- 4 otherwise.

Since the worst-case value of m is 4 in both cases, it follows from Table 4.3 that six cycles is a safe overapproximation for MLA and MUL, while seven cycles covers SMLAL, SMULL, UMLAL and UMULL.

4.2 Pipeline Stalls

As commented for the LDR instruction in Table 4.3, the pipeline in the ARM9TDMI processor core stalls in certain situations, where registers are loaded with contents from memory. The stalls give rise to extra cycles that are spent waiting for data to become available, and their effects must therefore be captured in order to determine safe WCETs. The stall situations are not exhaustively documented by ARM Ltd., but [27, Section 7.2] provides four examples. In this section we go over the details of these examples. The examples assume that the fetch and memory stages are able to load an instruction and a data element, respectively, in one cycle.

The first example consists of the following code sequence, which loads register R0 with the contents of the memory cell pointed to by R1 and subsequently saves the sum of R0 and R1 in R2:

```
LDR R0, [R1]
ADD R2, R0, R1
```

Since the ADD instruction cannot proceed to the execute stage before R0 has been loaded by LDR in the memory stage, ADD stalls in the decode stage for one cycle. The flow of the code sequence through the pipeline stages is illustrated in Table 4.4.

The second example loads register R0 with the one byte memory contents at the address determined by adding one to value of R1:

```
LDRB R0, [R1, #1]
ADD R2, R0, R1
```

After the load, the same addition instruction as in the previous example is carried out. Because the load instruction in this example requires a bitwise rotation, which happens in the writeback stage, ADD stalls for two cycles in the decode stage. The situation is illustrated in Table 4.5.

The third example loads registers R1, R2 and R3 with the three continuous words of memory contents pointed to by R12:

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	LDR				
2	ADD	LDR			
3		ADD	LDR		
4		ADD		LDR	
5			ADD		LDR
6				ADD	
7					ADD

Table 4.4: The flow of the code sequence `LDR R0, [R1]; ADD R2, R0, R1` through the pipeline stages. The `ADD` instruction stalls for one cycle in the decode stage, which is marked on the right side of the table.

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	LDRB				
2	ADD	LDRB			
3		ADD	LDRB		
4		ADD		LDRB	
5		ADD			LDRB
6			ADD		
7				ADD	
8					ADD

Table 4.5: The flow of the code sequence `LDRB R0, [R1, #1]; ADD R2, R0, R1` through the pipeline stages. The `ADD` instruction stalls for two cycles in the decode stage.

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	LDM				
2	ADD	LDM			
3		ADD	LDM		
4		ADD		LDM	
5		ADD		LDM	
6			ADD	LDM	
7				ADD	LDM
8					ADD

Table 4.6: The flow of the code sequence `LDM R12, {R1-R3}; ADD R2, R2, R1` through the pipeline stages. The `ADD` instruction stalls for two cycles in the decode stage.

Cycle	Fetch	Decode	Execute	Memory	Writeback
1	LDM				
2	ADD	LDM			
3		ADD	LDM		
4		ADD		LDM	
5		ADD		LDM	
6		ADD		LDM	
7			ADD		LDM
8				ADD	
9					ADD

Table 4.7: The flow of the code sequence `LDM R12, {R1-R3}; ADD R4, R3, R1` through the pipeline stages. The `ADD` instruction stalls for three cycles in the decode stage.

```
LDM R12, {R1-R3}
ADD R2, R2, R1
```

After the load, the sum of `R2` and `R1` is saved in `R2`. The `LDM` instruction stays in the memory stage for three cycles, because it needs to load three words. The loads are performed in ascending order according to the register numbers, i.e. in the order `R1`, `R2`, `R3`. The `ADD` instruction only reads `R2` and `R1` and are therefore allowed to enter the execute stage when `LDM` starts its final load in the memory stage. In other words, `ADD` stalls for two cycles in the decode stage. Table 4.6 illustrates the situation.

The fourth, final example is a variation of the third:

```
LDM R12, {R1-R3}
ADD R4, R3, R1
```

Since `ADD` now reads `R3` instead of `R2`, it must stall in the decode stage for three cycles instead of two. Table 4.7 illustrates the situation.

Chapter 5

The METAMOC Method

The purpose of this chapter is to present the METAMOC method — the Modular Execution Time Analysis using Model Checking method. An overview of the method is provided in Figure 5.1 on the following page. The information in the top row are the necessary inputs to the method. Typically, the only user-provided input is a binary executable, as the three other inputs are made available by researchers, hardware vendors or advanced users. A binary executable in this case is a file containing data that are ready to be executed directly on the intended hardware platform. If annotated source code is available, the method can extract loop bounds from it, however, loop bounds can also be input to the method manually.

The other inputs for the method are a model for the hardware's pipeline, a specification for the hardware's caches, and a model for the hardware's main memory. If the hardware platform is changed, these inputs might need adaptation. The box in the bottom of the figure represents the output in the form of a WCET of the executable given as input.

Cache and main memory in most hardware platforms can be parameterised in a common way, as they work according to the same principles. For most systems — especially embedded systems, as they often use simple main memories — the cache differs only in the parameters associativity, size, policies and speed, while the main memory differs only in size and speed. The method utilises these circumstances by taking cache parameters as input rather than concrete UPPAAL models. Using the parameters, the method adapts general UPPAAL models for the caches automatically.

The control flow of the process is reconstructed from the assembler code and a path model is created based on the control flow. A CFG is constructed for each function in the process. Many types of models can be used for the CFGs, however, the type considered by the METAMOC method are UPPAAL timed automata, as introduced in Section 2.3 on page 15. Each transition in the automata performs an abstract execution of an assembler instruction by feeding it to the automata that resemble the hardware's pipeline. Function calls and returns are accomplished by having the automata "transfer" control between each other using synchronisation channels.

Since the contents of processor registers are not tracked, the memory addresses accessed by the majority of the assembler code's load and store instructions are not known. It is essential to deal with this problem, as the performance

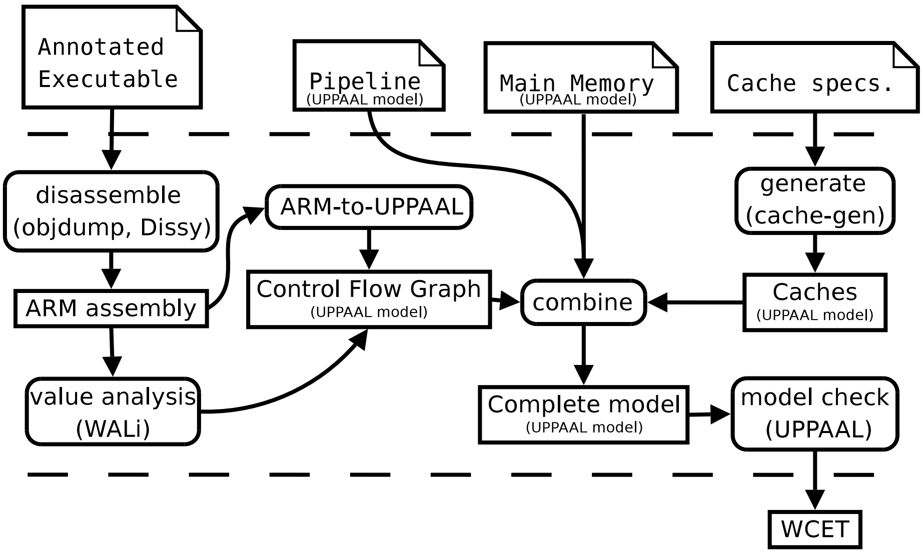


Figure 5.1: Overview of the METAMOC method. Rounded boxes represent utilities, squared boxes represent information, and edges represent transfer of information.

speed-up from caches cannot be guaranteed without knowledge of the concrete addresses. The method attempts to determine the memory addresses by conducting a value analysis on the assembler code. Using the formalism of WPDS, the value analysis finds an overapproximation of the possible register values at each execution point by evaluating the effects of the process' instructions. The results from the value analysis are added to the automata for the process' functions.

The automata for the process' functions are combined with automata for pipeline stages, caches and main memory, resulting in a NTA. Each automaton contributes a set of local and a set of global declarations, where the latter sets are combined with the sets from the other automata. Also, all automata are instantiated in the NTA's system declaration. The global clock `cyclecounter` is added in order to measure the number of processor cycles that passes during the abstract execution done by exploring the NTA. The final step of the method performs a full state space exploration of the NTA, determining the largest possible value for `cyclecounter`. The determined value is an overapproximation of the WCET of the process when executed on the chosen hardware.

Using NTAs as the modelling formalism allows for a loosely coupled model, which yields a flexible method where parts can easily be replaced or reused. For instance, the model of the hardware platform can be separated from the model of the process, and a process model can be combined with models of different hardware platforms.

The task of finding WCETs is often composed of four analyses: cache analysis, value analysis, pipeline analysis and path analysis [20]. We have followed this pattern for METAMOC, since it provides a modular separation. The characteristics of the analyses are explained below:

The cache analysis is responsible for taking the cache into account when performing WCET analysis. It does so by finding the set of memory blocks that are in the cache, or not, at all execution points of a process. As a process normally takes some form of input, it might be impossible to know whether or not a memory block is in the cache or not. The typical approach is therefore to make an underapproximation of the set of memory blocks which must be in the cache at some point. This approach is safe as long as the hardware platform does not exhibit timing anomalies. To find a safe underapproximation of the set of memory blocks in the cache, a description of the behaviour of the cache is needed. Often the hardware platform contains both instruction and data caches and might even have several levels of caches. In order to provide sharp WCETs, the cache analysis should model this.

The value analysis determines information about the value of registers and variables which are otherwise only available through an execution of the process. Not all values can be determined by a value analysis — for instance, input to the process introduces an element of uncertainty. A basic example is dereferencing a pointer: to know which memory address is accessed, the value of the pointer must be known. Value analyses approximate this information by giving a potential superset of the values the pointer can have. A value analysis can also be used to determine bounds on loops. In some cases loop bounds cannot be detected due to e.g. complex logic expressions and must be annotated manually by the programmer.

The pipeline analysis models the behaviour of the processor core's data path. This includes modelling processor features such as pipelining, delay slots, branch prediction, out-of-order execution and speculative execution. In case of missing information from other analyses or as a consequence of input, a precise model of the processor core might include some degree of non-determinism. The non-determinism, however, might make the state space exploration done by model checking intractable. One way to solve this problem is to abstract the non-determinism away by making safe assumptions about the hardware behaviour. This is not possible if the processor exhibits timing anomalies. In addition, making these assumptions of course conflicts with the desire for obtaining as sharp as possible WCETs.

The path analysis is used to find the worst execution path (in terms of time usage) through the control flow of the process. This requires the control flow of the process to be reconstructed from the machine code. This involves a number of challenges, on both high- and low-level representations of the process. This includes recovering enough flow information that has been lost during compilation, handling function calls, recursive functions, jumps to static addresses, conditional static jumps, jumps to addresses determined dynamically at runtime, etc.

The following four sections explain how the four analyses are handled in the METAMOC method. In the fifth, final section, the implementation of a graphical front-end to easily perform the analyses in a user-friendly manner.

5.1 Cache Analysis

The purpose of the cache analysis is to take the effects of caching into account in a safe, and as accurate as possible, way. This is done by determining cache hits in as many execution points in a process as possible, such that the penalty of accessing main memory is paid in as few execution points as possible.

The cache analysis for the ARM920T has been divided into two analyses: an instruction cache analysis and a data cache analysis. The analyses should model the impact of the respective cache on the WCET of the process in a safe way. In other words, the impact must be modelled pessimistically. The simplest and most pessimistic model is a cache model which always assumes that an access to a memory block results in a cache miss. The analyses are performed by modelling each cache as a timed automaton which communicates with a timed automaton modelling the main memory. In this way, if both the instruction and the data cache tries to access main memory, the first access will block the other.

Two different cache models have been made. In the first model, every memory access results in a cache miss, whereas in the second model, a concrete cache is imitated. The second model stores the addresses of the memory blocks that are currently in the cache. Modelling-wise, the first model is a special case of the second with some functionality removed. Therefore only the second cache model, depicted in Figure 5.2 on the facing page, is explained in detail. The advantage of the first model is that it will result in a much smaller state, meaning a larger state space can be explored by the model checker. It will, however, also result in WCETs which are not as sharp as those found using the other cache model. A combination of the two models — where the second model is used for the instruction cache and the first model is used for the data cache – might provide a good trade-off between the size of the state space and the sharpness of the WCETs.

A third approach could be to precompute the cache behaviour, given some context, and input this as a simple model, with no state. For example, the behaviour could be precomputed using abstract interpretation on abstract cache states, as described in Section 3.4.1 on page 36.

The model in Figure 5.2 on the facing page shows a timed automaton modelling the instruction cache. The UPPAAL declarations for the used functions are available in Appendix A on page 101. A similar timed automaton exists for the data cache. The exact differences will be listed after a detailed explanation of the automaton in Figure 5.2. The automaton can be divided into three parts: initialisation of the cache, cache writes, and cache reads. The initialisation starts in the initial location and is forced to take the outgoing transition by the synchronisation on the urgent broadcast channel `initCaches`. This way the initialisation can only happen once and it will end up in location `A`. In location `A`, the cache model is able to synchronise on either of the urgent channels `instructionCacheWrite` or `instructionCacheRead`, each representing the write and read part, respectively. This will imitate a write or read to the address passed to the model in the variable `instrAdr`. In both parts, three functions are used: `cache_contents`, `insert` and `update`. The function `cache_contents` is used to check if the instruction, with the address `instrAdr`, is in the cache. It returns minus one if the instruction is not in the cache, otherwise it returns the number of the cache line containing the instruction. If an instruction is not

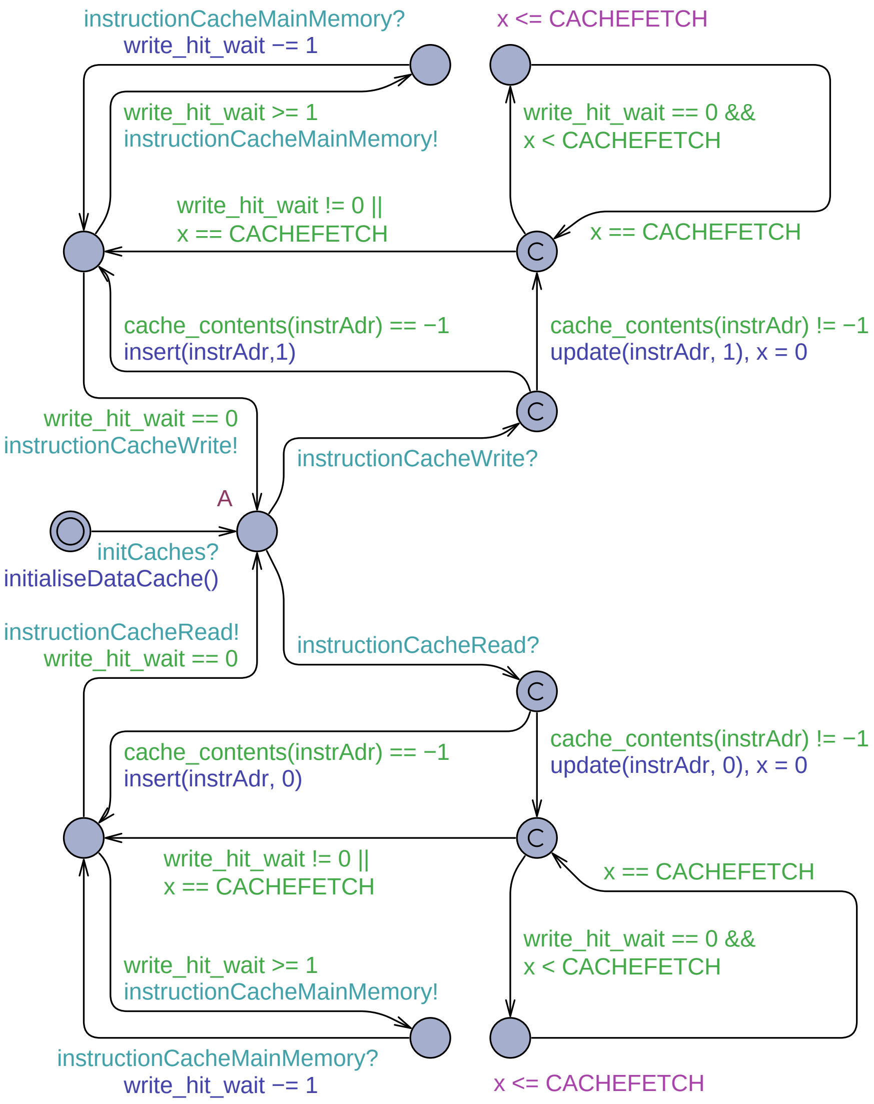


Figure 5.2: The timed automaton modelling the instruction cache.

in the cache, a cache miss is imitated. This is done using the insert function. If the instruction is already in the cache, a cache hit is imitated. This is done using the function update.

Both insert and update set the variable write_hit_wait to the number of memory accesses that the cache miss/hit will result in. For instance, a write to an unknown memory block might cause two memory accesses, if the cache has a write back/write allocate policy. Firstly, the memory block will be allocated, causing one memory access, and secondly, the memory block that is evicted might be dirty, causing a second memory access.

Since caches are slower than registers, even in case of a cache hit which does not result in a memory access, it will take some constant amount of time to access the cache. This is modelled by forcing both the read and the write part of the model to delay this amount of time. The amount of time is specified by the constant CACHEFETCH. If the variable write_hit_wait is equal to zero, which it is in case of a cache hit, the access does not result in a memory access.

The last steps of both the read and the write part imitate the correct number of memory accesses and finally synchronise back using the urgent channel instructionCacheWrite and instructionCacheRead, respectively. The synchronisation transition returns back to location A.

The value of write_hit_wait, when the insert or update function is called, is determined by the cache replacement policy, as the replacement policy is responsible for choosing which memory block to evict. Therefore, the replacement policies presented in Section 3.1 on page 23 have been implemented. The implementation is divided into two parts like the definition of the replacement policies: one handling cache hits and one handling cache misses. Each part is implemented as a function. For FIFO, the two relevant functions are named cacheHitFIFO and cacheMissFIFO. The functions for LRU follow the same naming scheme. Depending on the replacement policy of the cache, insert and update call cacheMissFIFO or cacheMissLRU.

The data cache analysis is similar to the instruction cache analysis. In the model used for the data cache analysis, the names of the synchronisation channels instructionCacheWrite and instructionCacheRead are renamed to dataCacheWrite and dataCacheRead, respectively. Also, the channel instructionCacheMainMemory, which is used to synchronise with the main memory model, is renamed to dataCacheMainMemory.

The timed automaton for main memory only delays for a constant amount of time on each access. The automaton is available in Appendix B on page 105. One could imagine memory models more complicated, such as a model of SDRAM, where the access time is not constant but depends on the internal state of the memory.

As described in Section 3.1 on page 23, there are several parameters for the caches. To ease the creation of cache models, a tool has been developed to generate cache models. The tool takes a specification of the cache and a general cache model as input. The specification contains the following information:

- The cache type (data or instruction).
- The number of cache lines.
- The number of cache sets.

- The write policies (“write allocate” or “no write allocate”, and “write through” or “write back”).

- The replacement policy (one of the replacement policies mentioned in Section 3.1 on page 23).

The cache generation tool does not currently support multiple levels of caches as well as unified caches and combinations thereof. In the future, support for such hardware could be added, along with support for more replacement policies.

As mentioned in Chapter 4 on page 41, the ARM920T processor supports two replacement policies: FIFO and pseudo-random [28, p. 44]. We have only implemented support for the FIFO replacement policy.

The reason for not implementing the pseudo-random replacement policy is the fact that after a single access to an unknown memory block, all information about the cache is lost. To illustrate this, an example is given in Figure 5.3 on the next page, showing an unknown memory block being accessed, and because of the pseudo-random replacement policy, the memory block can be placed in any of the cache lines. This means that the analysis must assume an empty cache from this point.

According to [20], to be able to make precise statements about timing behaviour, an important processor property is that the cache replacement policy should be immune to “chaos”. For instance, if the value analysis is not able to give precise information about which memory block will be accessed, this can cause the replacement policy to lose information. If the replacement policy can recover information, it is immune to “chaos”.

The pseudo-random replacement policy is not immune to “chaos”. In Figure 5.3 an example is given, where the Pseudo-random replacement policy is not able to recover knowledge. The figure shows two abstract cache states. In the state $\hat{c}$, four memory blocks are known to be in the cache and the last memory access was m_4. The abstract cache state $\hat{c}'$ shows the state after a memory access to an unknown memory block. This entails that it is unknown whether the memory access will result in a cache hit or a cache miss on the abstract cache state $\hat{c}$. What can be seen in part (b) of Figure 5.3, is that we have to assume the worst, which means a cache miss. Since the memory address is not known, it cannot be safely determined which of the four cache sets or which of the two ways the memory block should be placed in. Based on this knowledge, we have choosen not to implement the pseudo-random replacement policy.

5.2 Value Analysis

The value analysis is responsible for finding an overapproximation of the values each register can have at a given execution point in the process (possibly given some context), for use in determining which memory address is accessed at a certain point in the execution.

We have chosen to base our value analysis on the framework of WPDSs as described in Section 2.4 on page 17. We have chosen WPDSs, even though our analysis is quite naïve and simple, because it allows us to benefit from advances within the WPDS framework, and it allows us to increase the precision of the analysis later on, e.g. by adding more context-sensitivity.

(a)

Abstract cache state $\hat{\mathbf{c}}$	$\mathbf{s_1}$	$\mathbf{s_2}$	$\mathbf{s_3}$	$\mathbf{s_4}$
way 1	$\{m_1\}$			$\{m_4\}$
way 2		$\{m_6\}$	$\{m_7\}$	

$\Downarrow$ access to unknown memory block

(b)

Abstract cache state $\hat{\mathbf{c}}'$	$\mathbf{s_1}$	$\mathbf{s_2}$	$\mathbf{s_3}$	$\mathbf{s_4}$
way 1				
way 2				

Figure 5.3: Abstract cache states for a 2-way set associative cache with pseudo-random as replacement policy, showing the replacement policy is not immune to "chaos".

We have desined and implemented a relatively simple value analysis. The analysis is first and foremost an sound[1], easy to implement (implementable within the timeframe of the project), and finally, as precise and fast as possible. We have made a couple of choices to satisfy these requirements:

- Our intra-procedural analysis is flow-sensitive, and we unroll loops according to their loop bound. Our inter-procedural analysis is also flow-sensitive, and as precise as allowed by the WPDS formalism, i.e. function calls return to their call site.

- We are totally data-insensitive, with regards to the control flow. An example is if a conditional branch exists, as in this program fragment:

  ```
  1  if (x > 5) {
  2    a();
  3  } else {
  4    b();
  5  }
  ```

  In this case, when the execution reaches line one, we assume it can flow to either line two or line four, regardless of what the value analysis might have infered about the value of x at line one.

- We only track values in registers, not values in main memory. This was primarily chosen to simplify the implementation. This loses quite a lot of precision, as any load from memory results in an unknown value. Note that just tracking the contents of the stack should result in quite an improvement of the precision.

- A register can have a single, known value, otherwise the value is unknown. This means if there are two paths to a combine-point, and the two paths have different values for a register, the combined result has that register as unknown.

- To simplify the analysis implementation greatly, the implementation traces syntactic values only. The implementation does not interpret any of the

[1] That is, the analysis never underapproximates the set of values.

values; this is done by post-processing the results and interpreting the formulae.

The advantage of this approach is that the implementation only needs to implement a single operation: syntactic substitution. The disadvantage is that the formulae can become quite large[2], so to be efficient, a limit to the length of the formulae must be set, such that if a formula grows beyond that limit, the register's content is assumed to be unknown. This is sound, as all values is a very coarse overapproximation of the one value the formula described.

In our case we estimate the lost precision to be negligible, as we are only concerned with values used as pointers to memory, and these tend to not be used in calculations continuously.

There are two major components to our use of WPDSs: the weight domain and the construction of the PDS. We first describe our weight domain, and then describe how we construct the PDS.

5.2.1 Weight Domain

Our weight domain basically describes the effects that instructions, and sequences of instructions, can have on the registers' values.

We will actually describe two weight domains. Firstly, a more abstract weight domain, which conveys the underlying principles very well, but would require an implementation to interpret values semantically. Secondly, we describe a less abstract weight domain, i.e. one that uses only syntactic substitution. The latter weight domain is the one we have actually implemented. We prove no relationship between the two weight domains, but note that they are closely related.

Definition 15. *The weight domain REGEFFECT* $= (D, \oplus, \otimes, \bar{0}, \bar{1})$ *is defined as:*

D $= ((\mathbb{N} \cup \{\top, \bot\})^{|REGS|} \to (\mathbb{N} \cup \{\top, \bot\})^{|REGS|})$, *where* $REGS = \{r_0, \dots r_n\}$ *is the set of registers in the architecture. Thus, our domain is the set of environment transformers that transform an environment (a vector of values assigned to each register) into another environment. A special value, which a register can have, is the value* $\top$*, indicating that the register can have any value — its value is unknown. Another special value is* $\bot$*, which means that no information is known about this register — it is undefined. An example of a weight could be the weight representing the effect* $r_1 = r_0 + 2$*:*

$$f \begin{pmatrix} r_0 \\ r_1 \\ r_2 \\ \vdots \\ r_n \end{pmatrix} = \begin{pmatrix} r_0 \\ r_0 + 2 \\ r_2 \\ \vdots \\ r_n \end{pmatrix},$$

[2] During development we encountered an example that calculated the 30th Fibonacci number by addition, leading to the calculations $F_3 = 1 + 1, F_4 = (1 + 1) + (1 + 1), F_5 = ((1 + 1) + (1 + 1)) + \dots$. Continued syntactic substitution of these terms leads to a formula with length proportional to the value of the number, in this case 832040, leading to strings with lengths of a couple of megabytes.

signifying that r_1's value is changed, and all the other registers values remain unchanged. Notice that we use vector notation to better emphasise the changes done to individual registers. For a weight $f(r_0, \dots, r_n) = (e_0, \dots, e_n)$ we denote a component of f as $f_i(r_0, \dots, r_n) = e_i$, where $i \in \{0, \dots, n\}$.

$\oplus$ **(combine)** *is defined by:*

$$\forall a, b \in D, \forall i \in \{0, \dots, n\} : (a \oplus b)_i(x) = \begin{cases} a_i(x) & \textit{if } b_i(x) = \bot \\ b_i(x) & \textit{if } a_i(x) = \bot \\ a_i(x) & \textit{if } a_i(x) = b_i(x) \\ \top & \textit{otherwise.} \end{cases}$$

$\otimes$ **(extend)** *is reverse function application, such that:*

$$\forall a, b \in D, \forall i \in \{0, \dots, n\} : (a \otimes b)_i(x) =$$
$$\begin{cases} \bot & \textit{if } a_i(x) = \bot \vee b_i(x) = \bot \\ b_i(a_i(x)) & \textit{otherwise.} \end{cases}$$

$\bar{0}$ *is the bottom element, that is*

$$\bar{0} = \begin{pmatrix} r_0 \\ \vdots \\ r_n \end{pmatrix} = \begin{pmatrix} \bot \\ \vdots \\ \bot \end{pmatrix}.$$

$\bar{1}$ *is the identity relation, that is*

$$\bar{1} \begin{pmatrix} r_0 \\ \vdots \\ r_n \end{pmatrix} = \begin{pmatrix} r_0 \\ \vdots \\ r_n \end{pmatrix}.$$

To give an example of the weight domain, we use it to analyse the effect of the program fragment in Figure 5.4.

```
r0 = 5
r1 = r0 + 2
```

Figure 5.4: An example program fragment.

The weights associated with each line, assuming $REGS = \{r_0, r_1\}$ for brevity, would be: $v \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} 5 \\ r_1 \end{pmatrix}$ and $w \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} r_0 \\ r_0 + 2 \end{pmatrix}$, respectively.

Using the extend operation we can compute the effect of the entire program fragment, assuming the registers have the inital value of 0:

$$(v \otimes w) \begin{pmatrix} 0 \\ 0 \end{pmatrix} = w \left(v \begin{pmatrix} 0 \\ 0 \end{pmatrix} \right) = \begin{pmatrix} 5 \\ 5 + 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix}.$$

The combine operation can be demonstrated as follows. If

$$v \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} 5 \\ 7 \end{pmatrix} \quad \text{and} \quad w \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} 3 \\ 7 \end{pmatrix},$$

then

$$(v \oplus w) \begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \top \\ 7 \end{pmatrix}.$$

Definition 16. *The weight domain $REGEFFECT_{SYNTAX} = (D, \oplus, \otimes, \bar{0}, \bar{1})$ is defined as:*

D $=$ *$((form(REGS) \cup \{\top, \bot, \mathbf{id}\}) \rightarrow (form(REGS) \cup \{\top, \bot, \mathbf{id}\}))$, where REGS is the set of registers in the architecture, and $form(REGS)$ is the set of all formulae involving natural numbers, variables from the set REGS and the normal arithmetic operations (such as plus, minus, times, ...). Special values are the values $\top$, $\bot$ and **id**, which respectively indicate an unknown value, an undefined value and an unchanged value. The syntax for $form(REGS)$ can be seen in Figure 5.5.*

```
form(REGS)  :=  r  |  n  |
                form(REGS)  OPERATOR  form(REGS)  |
                '( '  form(REGS)  ') '
where  r ∈ REGS  and  n ∈ ℕ

OPERATOR  :=  +  |  −  |  ∗  |  /  |  >>  |  <<
```

Figure 5.5: The syntax for formulae in the $REGEFFECT_{SYNTAX}$ weight domain.

$\oplus$ *is defined by:*

$$\forall a, b \in D, \forall i \in \{0, \ldots, n\} : (a \oplus b)_i(x) = \begin{cases} a_i(x) & \text{if } b_i(x) = \bot \\ b_i(x) & \text{if } a_i(x) = \bot \\ a_i(x) & \text{if } a_i(x) == b_i(x) \\ \top & \text{otherwise} \end{cases}$$

where $==$ is syntactic equality.

$\otimes$ *is syntactic substitution, with a few special cases, defined as* $\forall a, b \in D, \forall i \in \{0, \ldots, n\} : (a \otimes b)_i(x) =$

$$\begin{cases} \bot & \text{if } a_i(x) = \bot \vee b_i(x) = \bot \vee \\ & \quad \exists j \in \{0, \ldots, n\} : (a_j(x) = \bot \wedge \\ & \quad j \in uses(b_i(x))) \\ \top & \text{if } b_i(x) = \top \vee \exists j \in \{0, \ldots, n\} : \\ & \quad (a_j(x) = \top \wedge j \in uses(b_i(x))) \\ a_i(x) & \text{if } b_i(x) = \mathbf{id} \\ b_i(x)[\forall j \in \{0, \ldots, n\} : & \text{if } b_i(x) \notin \{\top, \bot, \mathbf{id}\} \wedge \\ \quad r_j / \text{“(”} \circ a_j(x) \circ \text{“)”}] & \quad \forall j \in uses(b_i(x)) : a_j(x) \notin \{\top, \bot\} \end{cases}$$

where $uses(e)$ is a set containing the numbers of the registers used in formula e, $\circ$ is string concatenation, and $s[x/x', y/y', \ldots]$ is string substitution such that $s[x/x', y/y', \ldots]$ is equal to s, but with all occurrences of x replaced by x', occurrences of y replaced by y', and so forth. An important distinction is that the replacement happens in parallel, such that “r0”[“r0”/“r1”, “r1”/“r2”] = “r1”, and not “r2” as it would have been if the replacement was sequential.

The first case ensures that $\bar{0}$ is an annihilator with regards to the $\otimes$ operator, as required by the definition of a weight domain — that is, if a formula uses an undefined value, it itself becomes undefined.

The second case ensures that if an unknown value is used in a formula, the result is also unknown.

The third case handles the case that register x's value is unchanged by the right-hand side (b).

The fourth case is the syntactic substitution, such that if the right-hand side (b) has a formula for register x, the register names used in that formula are replaced with their values from the left-hand side (a) enclosed within parenthesis (to ensure that operator precedence is preserved).

$\bar{0}$ *is defined as:*

$$\bar{0}\begin{pmatrix} r_0 \\ \vdots \\ r_n \end{pmatrix} = \begin{pmatrix} \perp \\ \vdots \\ \perp \end{pmatrix}.$$

$\bar{1}$ *is defined as:*

$$\bar{1}\begin{pmatrix} r_0 \\ \vdots \\ r_n \end{pmatrix} = \begin{pmatrix} \mathbf{id} \\ \vdots \\ \mathbf{id} \end{pmatrix}.$$

Again, the usage of the weight domain $REGEFFECT_{SYNTAX}$ is demonstrated on the program fragment from Figure 5.4 on page 60.

The weights associated with each line, assuming $REGS = \{r_0, r_1\}$ for brevity, would be: $v\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \text{“5”} \\ \mathbf{id} \end{pmatrix}$ and $w\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \mathbf{id} \\ \text{“}r_0 + 2\text{”} \end{pmatrix}$, respectively.

Using the extend operation we can compute the effect of the entire program fragment, assuming the registers have the inital value of 0:

$$(v \otimes w)\begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \text{“5”} \\ \text{“}r_0 + 2\text{”}[r_0/\text{“(”} \circ \text{“5”} \circ \text{“)”}, r_1/\mathbf{id}] \end{pmatrix} = \begin{pmatrix} \text{“5”} \\ \text{“}(5) + 2\text{”} \end{pmatrix}.$$

The combine operation can be demonstrated as follows. If

$$v\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \text{“}(3) + 2\text{”} \\ \text{“7”} \end{pmatrix} \quad \text{and} \quad w\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \text{“}(3) + 2\text{”} \\ \text{“}(5) + 2\text{”} \end{pmatrix},$$

then

$$(v \oplus w)\begin{pmatrix} r_0 \\ r_1 \end{pmatrix} = \begin{pmatrix} \text{“}(3) + 2\text{”} \\ \top \end{pmatrix}.$$

Note that even though “7” is equal to “(5) + 2” semantically, this information is lost because this weight domain is purely syntactical. The use of syntactic equality in this context is still safe, even though we lose precision in the cases where the same value is calculated in two different ways. The reason this is safe, is that the combine operation gives the value $\top$, which, while unprecise, is still an overapproximation of a single value.

We omit a number of things in our use of WPDSs: we do not prove that our weight domains actually obey the restrictions set forth for weight domains. We do not prove that the weight domains give sound results, given the formal semantics of the ARM assembly language. The reason for this omission is primarily one of practicality; we have focused our effort on value analysis on getting a working implementation, that proves the concept. The value analysis is, seen in the bigger picture of METAMOC, an utility that is certainly necessary, but not at the core of the METAMOC method.

5.2.2 Construction of the Weighted Push-Down System

The WPDS is basically constructed as described in [36], i.e. using one control state and using the stack to keep track of the current state of execution. We make the assumption that the control flow is statically derivable, that is: "function calls" and "function returns" are easily identifiable.

There are basically three forms of transitions in the WPDS:

Sequential transitions: Sequential flow from one instruction to the next. Handled by a WPDS rule of the form $\langle p, j\rangle \hookrightarrow \langle p, k\rangle$ where j is the instruction address control can flow from, and k the instruction address control can flow to.

Function calls: Function calls are handled by pushing the call site on the stack, such that when the function returns, the execution will continue from there. The WPDS rule corresponding to a function call at instruction address j which is followed sequentially by instruction address k, to the function identified by *funcname*, is: $\langle p, j\rangle \hookrightarrow \langle p, f_{funcname}\ k\rangle$. Rules for each function exist, such that $\langle p, f_{funcname}\rangle$ $\hookrightarrow \langle p, i\rangle$, where i is the address of the first instruction in the *funcname* function.

Function returns: Function returns are handled by a return rule that simply pops from the stack. If the address of the return is j, the rule is: $\langle p, j\rangle \hookrightarrow$ $\langle p, \epsilon\rangle$.

In this way the analysis is inter-procedurally flow-sensitive, as function calls return precisely to their call-site.

To improve the precision of the analysis, loops are unrolled such that the body of a loop is copied and appended sequentially as many times as the loop bound indicates. Loop unrolling is a technique originally used in compiler optimisation, which basically replaces the backward jump by duplicating the body of the loop. An example of loop unrolling can be seen in Figure 5.6. Loop unrolling provides additional precision in our analysis, because each instruction in the loop will then be tracked in the context of the loop counter.

```
for (i = 0; i < 5; i++) {
    r0 = r0 + 1;
}
```

```
r0 = r0 + 1
r0 = r0 + 1
r0 = r0 + 1
r0 = r0 + 1
r0 = r0 + 1
```

Figure 5.6: An example of loop unrolling.

Each transition in the WPDS is assigned a weight according to the effect of the instruction that the transition is from. A table of some example instruction types, and their associated weights, can be seen in Table 5.1 on the next page.

5.2.3 Implementation

Our implementation has been created using the open source WPDS library WALi [21]. WALi is written in C++, has a number of example weight domains,

Instruction type	Associated weight
`MOV R0, x`	$r_0 = x$
`MOV R0, R1`	$r_0 = r_1$
`ADD R0, R1, x`	$r_0 = r_1 + x$
`ADD R0, R1, R2`	$r_0 = r_1 + r_2$
`SUB R0, R1, x`	$r_0 = r_1 - x$
`SUB R0, R1, R2`	$r_0 = r_1 - r_2$
`LSL R0, R1, x`	$r_0 = r_1$ << x
`ASR R0, R1, x`	$r_0 = r_1$ >> x
`PUSH R0,...,Rn`	$sp = sp - n \cdot 4$
`POP R0,...,Rn`	$r_0 = \top; \ldots; r_n = \top; sp = sp + n \cdot 4$

Table 5.1: The weights associated with different instruction types. The variable `x` is a constant integer value, while `r0`, ..., `rn` can be any register names. The separator ";" implies that the value of more than one register is to be set. Registers, which are not mentioned in the weight, are implicitly assigned the value **id**.

and is easily extensible with new ones. We have implemented the weight domain $\mathit{REGEFFECT}_{\mathit{SYNTAX}}$ in C++.

Since most of our tools are written in Python, we would like to write as much as possible of the value analysis in Python. We have therefore developed Python bindings for the WALi library, allowing us to create WPDSs, including creating transitions and assigning weights. The bindings allow for a good performance/-functionality trade-off, as the performance sensitive parts (the weight domain operations and the WPDS algorithm) can be implemented in the performance optimised language C++, while the less performance-critical parts (constructing the WPDS) can be implemented in Python. In addition we gain the ability to reuse code, originally written for other purposes.

When WALi has computed an answer, the weights will have to be semantically evaluated. Fortunately[3], our weights are all syntactically correct Python expressions. We have therefore simply used the Python interpreter to evaluate the expressions and calculate the concrete values.

5.3 Pipeline Analysis

The purpose of the pipeline analysis is to take the pipeline's impact on the WCET into account. We imitate the parallel nature of the pipeline by modelling the pipeline stages as individual timed automata. The flow of instructions through the stages is achieved by having the automata synchronise with each other as illustrated in Figure 5.7 on the facing page. For each automaton information is stored about the instruction occupying it, and this information is passed on when the automaton synchronises with the next automaton. The information is stored globally, enabling the automata for the process functions to feed the fetch stage with instructions. In addition, the information also enables transitions in a stage to depend on instructions in the other stages, which is

[3] Actually, this is by design.

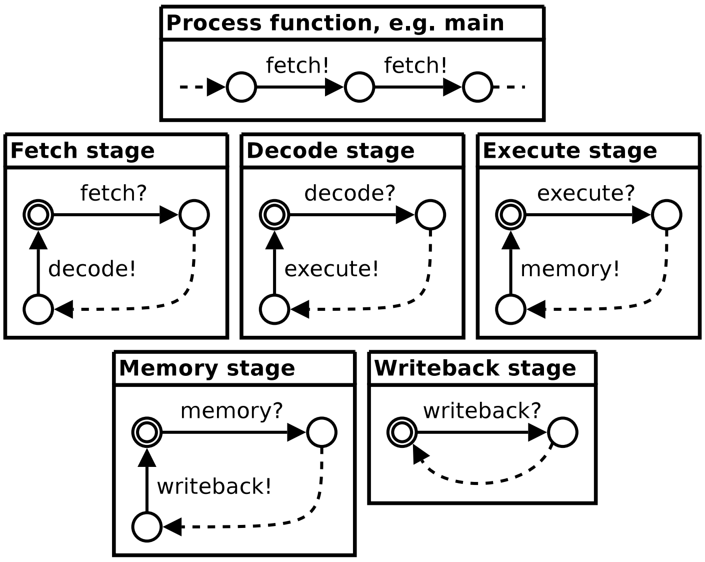


Figure 5.7: Overview of the synchronisation between the automata for process functions and the automata for pipeline stages.

necessary for implementing pipeline stalls. The information stored is the instruction address, the instruction type, the address of data accessed in main memory by the instruction, a bit mask for registers read and a bit mask for registers written.

As is clear from Figure 5.7, all stage automata are cyclic, as all of them end up in their initial location. This enables the pipeline to continue executing instructions as long as the function automata for the process being analysed are not done. Progress with limited or no time delay is obtained using invariants, committed locations and urgent channels.

Non-determinism is removed by having priorities on the stage automata, entailing that a transition in an automaton is enabled only if no transitions are enabled in higher priority automata. The priorities are assigned in increasing order, from the fetch stage automaton to the writeback stage automaton, such that the former has lowest priority and the latter has highest priority. The purpose of removing the non-deterministic interleaving of the pipeline stages is to reduce the state space and thereby reduce memory and time consumption of the state space exploration.

Recall from Section 2.3 on page 15 that the WCET is found by verifying the property `sup: cyclecounter`, which determines the supremum for the global clock `cyclecounter` during a full state space exploration. This approach requires that time must not pass when the NTA has reached its deliberate, ending deadlock. If time was allowed to pass, the clock would not be bounded, and UPPAAL would give the trivial guarantee `cyclecounter < infinity`.

The required time-bounding can be achieved by having at least one of the automata deadlock in a committed location or by having all automata deadlock in urgent locations [10]. The former scheme makes it easy to model the fact that the pipeline must be empty for the execution to be done. Figure 5.8

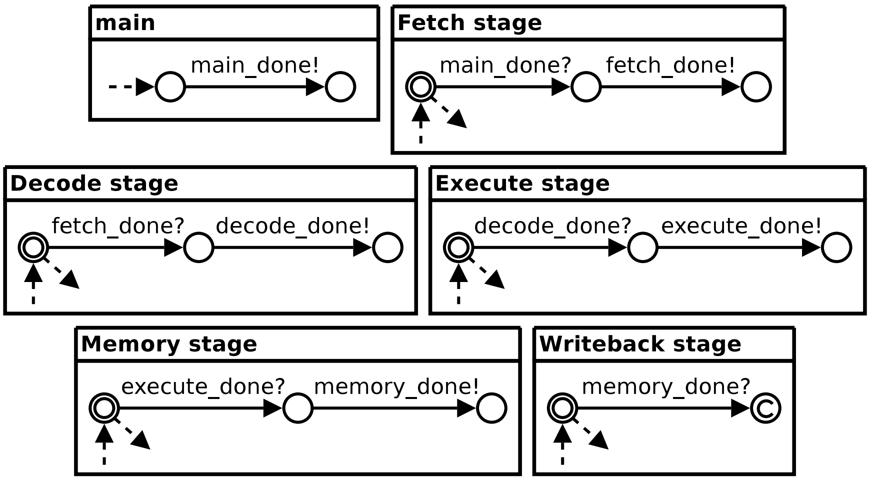


Figure 5.8: The chain of synchronisations ensuring that the NTA deadlocks in a committed state, thereby bounding the global clock `cyclecounter`.

illustrates how the scheme is implemented through cooperation between the automaton for the process' main function and the automata for the pipeline stages. A chain of synchronisations is initiated when the last transition in the main automaton is taken. Ultimately, the NTA deadlocks when the memory and writeback automata synchronise over the urgent channel `memory_done` and the committed location in the writeback automaton is entered.

Below the stages of the pipeline is presented in detail.

The fetch stage is the part of the pipeline where instructions are fetched from the instruction cache or main memory. The automaton for the stage is depicted in Figure 5.9. Using the channel `instructionCacheRead`, the automaton synchronises with the instruction cache automaton, which again synchronises with main memory if the requested instruction is not in the cache. The fetch automaton does not use any local clocks, as it is the responsibility of the instruction cache automaton and the main memory automaton to delay the appropriate number of cycles.

Since instructions are not evaluated before the execute or memory stages of the pipeline, the fetch and decode stages will contain two wrong instructions in the event that a branch is taken. The problem only exists in hardware, since the method presented here evaluates branch instructions using loop bounds in the CFG automata, i.e. before the instructions are fed to the pipeline. In hardware, the problem is solved by flushing the fetch and decode stages when a branch is taken. Even though the two flushed instructions will no longer be evaluated, they have been loaded into the instruction cache, and to get safe WCETs this side-effect must be modelled. As illustrated in Figure 5.10 on page 68, the fetch stage is able to distinguish the two possible outcomes of a conditional branch instruction, as the two outgoing transitions from a location have different instruction types. The event that a branch is taken has the type `INSTR_BRANCH`, while the opposite has the type `INSTR_OTHER`. Since the two flushed instructions always directly succeed the branch instruction in the

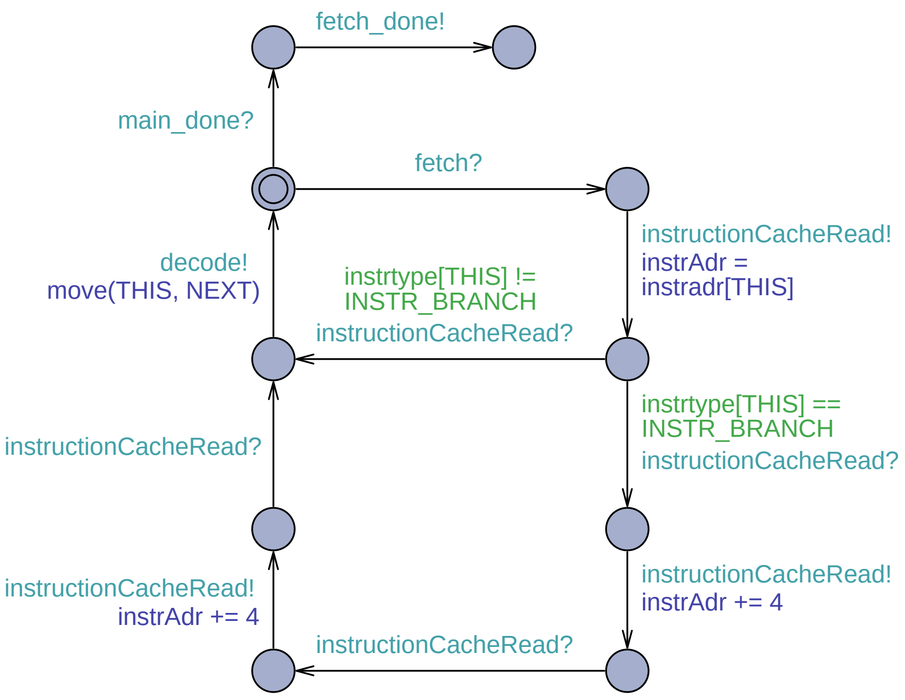


Figure 5.9: UPPAAL automaton for the pipeline's fetch stage.

instruction stream, the fetch automaton performs two extra fetches from offsets four and eight, relative to the address of the branch instruction. This special handling is done in the lower part of Figure 5.9. The chosen offsets come from the fact that each 32 bit ARM instruction takes up four bytes of memory.

Contrary to conditional branching, unconditional branch instructions have only a single outgoing transition, which is always marked with the `INSTR_BRANCH` instruction type. The reason being that these instructions almost always change the program counter more radically than sequential progression. In the code generated by GCC from the benchmark programs published by Mälerdalens WCET Research Group, unconditional branches are typically function calls, function returns, or jumps in connection with the C language `switch` construct.

The last transition in the fetch automaton's loop moves the instruction data to the decode stage and synchronises with the decode automaton. The function call `move(THIS, NEXT)` accomplishes the move, where `THIS` and `NEXT` are two constants for indexing the global information array. For example, in the fetch automaton the values are declared with the following values:

```
THIS = PIPELINE_FETCH_STAGE
NEXT = PIPELINE_DECODE_STAGE
```

In Figure 5.11, the automaton for the decode stage is depicted together with a small helper automaton. Contrary to real hardware, actual decoding of instructions in the decode automaton is not necessary, since the method is data insensitive. Instruction decoding is instead thought to happen while the

```
Address   Instruction

...
0x8370    MOV R1, R0
0x8374    BNE 834C
...
```

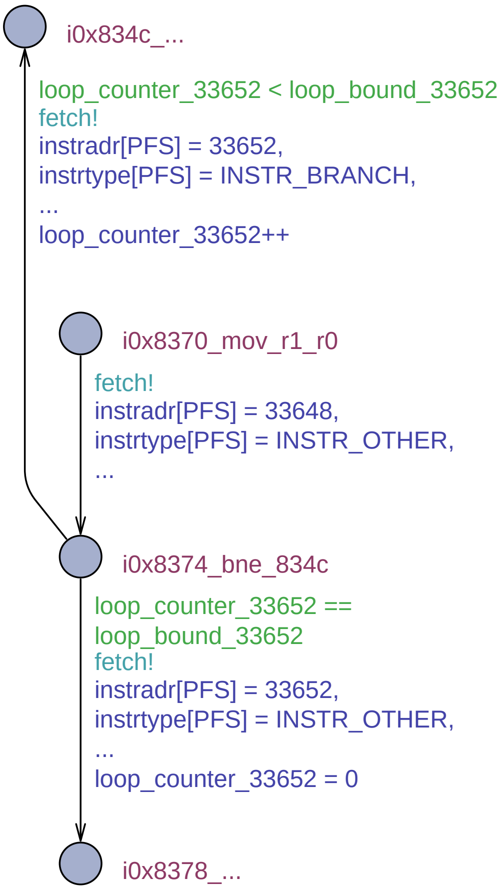


Figure 5.10: ARM assembly and the corresponding UPPAAL automaton. To make the model fit on this page, `PIPELINE_FETCH_STAGE` has been abbreviated as `PFS`. Due to the distinct instruction types on the outgoing transitions from the `i0x8374_bne_834c` location, the fetch stage is able to provoke additional instruction fetches in the event that the branch is taken.

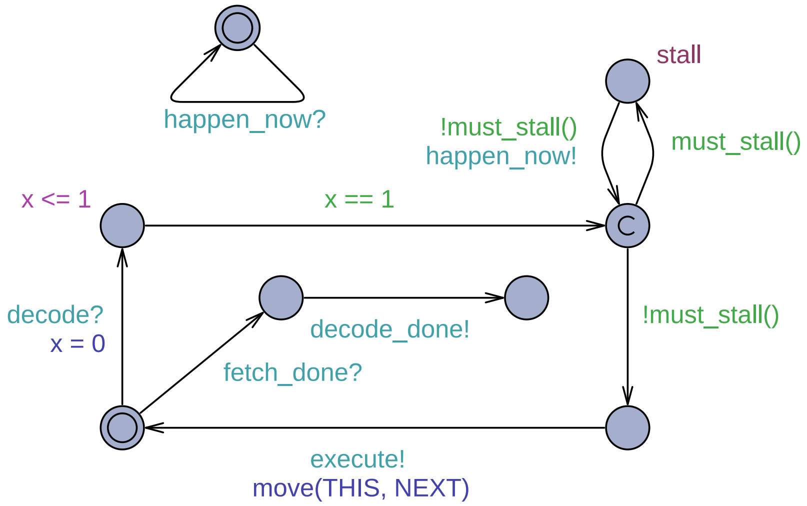


Figure 5.11: UPPAAL automaton for the pipeline's decode stage together with a small helper automaton to force progress in the decode automaton when it does not need to stall any longer.

automaton delays for one cycle in the upper left location. After the initial delay, the automaton determines whether it must stall or not. Recall that stalling means that the decode automaton must delay before synchronising with the execute automaton, because the instruction in the decode stage depends on the instruction in the memory or writeback stages. The function `must_stall` checks for this dependency:

```
bool must_stall()
{
        /* Check if we must wait for one or more registers
         * to be loaded in the memory stage. */
        if ((instrtype[PIPELINE_MEMORY_STAGE] == INSTR_POP ||
            instrtype[PIPELINE_MEMORY_STAGE] == INSTR_LOAD ||
            instrtype[PIPELINE_MEMORY_STAGE] ==
            INSTR_LOADROTATE) && (regread[THIS] &
            regwrite[PIPELINE_MEMORY_STAGE]) > 0)
                return true;

        /* Check if we must wait for a value to be rotated
         * in the writeback stage. */
        if (instrtype[PIPELINE_WRITEBACK_STAGE] ==
            INSTR_LOADROTATE && (regread[THIS] &
            regwrite[PIPELINE_WRITEBACK_STAGE]) > 0)
                return true;

        return false;
}
```

Using binary AND (&) on register masks, `must_stall` determines whether the instruction currently in the memory stage is a load instruction, writing to

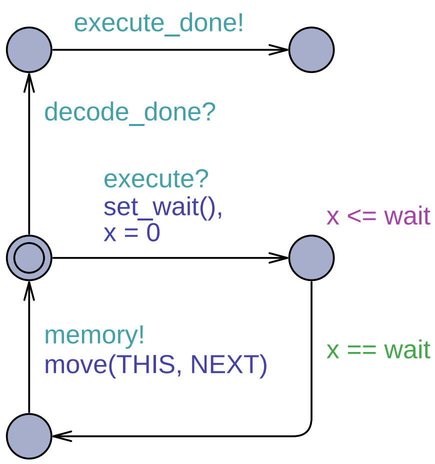


Figure 5.12: UPPAAL automaton for the pipeline's execute stage.

any registers that the instruction in the decode stage reads from. Similarly, if there is a rotation-requiring load instruction in the writeback stage, the function checks whether there is an overlap between the registers written by the load instruction and the registers read by the instruction in the decode stage. The reason for the second check is that the rotation is carried out in the writeback stage, and the result of the load is not ready before the rotation has been performed.

Since the memory automaton clears the bits in its register write mask continously as loads are performed, the decode automaton only stalls for a minimum amount of time. For example, if a `LDM` (load multiple) instruction in the memory stage loads the registers `R1`, `R2` and `R3` with values, and an `ADD` instruction in the decode stage needs the values in registers `R1` and `R2`, the decode automaton will stop stalling immediately when registers `R1` and `R2` — *but not yet* `R3` — have been loaded. This is similar to the third example in Section 4.2 on page 48.

The small helper automaton forces progress in the decode automaton, as it enables the automaton to have a synchronisation over the urgent channel `happen_now`. The effect is that the synchronisation transition is taken immediately when `must_stall` evaluates to false. Our first approach to achieve this was to stall for one cycle at a time, cycling around in a loop continously checking the stall condition. The approach did not need a helper automaton and worked in the sense that it made the model stall the correct number of cycles. Unfortunately, it created a greater state space than the current approach due to the many transitions, thereby lowering the size and complexity of the programs that the method was able to analyse. It was attempted to make the helper automaton in the current approach superfluous by removing the `happen_now!` synchronisation and adding the invariant `must_stall()` to the stall location. This change, however, causes UPPAAL to deadlock in an undefined state, as the instruction processing in the memory and writeback automata, which causes `must_stall()` to become false, leads to a violation of the invariant and is therefore not allowed. That was the reason for applying this little modelling trick.

Shifts and arithmetic operations are performed in the execute stage of the data path, which is depicted in Figure 5.12. Compared to the previous two stage automata, the execute automaton is quite simple. Since the method is data insensitive, the automaton only needs to delay for a number cycles corresponding to executing the instruction. To keep the automaton's contribution to the overall state space as small as possible, the function `set_wait` is used to set the integer `wait` to the appropriate number of cycles:

```
void set_wait()
{
     if (instrtype[THIS] == INSTR_MUL1)
          wait = CYCLES_MUL1;
     else if (instrtype[THIS] == INSTR_MUL2)
          wait = CYCLES_MUL2;
     else
          wait = CYCLES_OTHER;
}
```

The instruction type `INSTR_MUL1` covers the multiplication intructions `MLA` and `MUL`, whereas `INSTR_MUL2` covers `SMLAL`, `SMULL`, `UMLAL` and `UMULL`. The number of cycles needed to execute any of the multiplication instructions on the ARM9TDMI processor core depends on the operands [27, Section 7.1], and to handle this as simple as possible we have chosen two safe upper bounds for the constants `CYCLES_MUL1` and `CYCLES_MUL2`. The upper bounds are six and seven cycles, respectively. All other instructions are delayed for one cycle, since this is the number of cycles needed to do addition, subtraction, shifting, etc., and it is also the minimum number of cycles that any instruction must stay in the execute stage.

Figure 5.13 shows the automaton for the memory stage. Instructions for loading and storing data are handled in the memory stage. The automaton has three main paths: one for handling loads (stack popping, general loading from memory, and loading that requires rotation), one for handling stores (stack pushing and general storing to memory), and one for handling all other types of instructions. Instructions not relevant to the memory stage are handled by delaying for one cycle.

For load instructions, the memory automaton must issue a load from the data cache automaton for each destination register specified by the instruction, whereas for store instructions, it must issue a store for each source register. The load and store operations are initiated by having the two automata synchronise over the urgent channels `dataCacheRead` and `dataCacheWrite`, respectively. Recall that registers read by an instruction are marked in the register mask `regread[THIS]`, whereas registers that are written to are marked in `regwrite[THIS]`. The handling of a load or store instruction reduces to looping through the 16 bits in the relevant mask and synchronising with the data cache automaton for each set bit. The function `is_set` determines whether a particular bit is set:

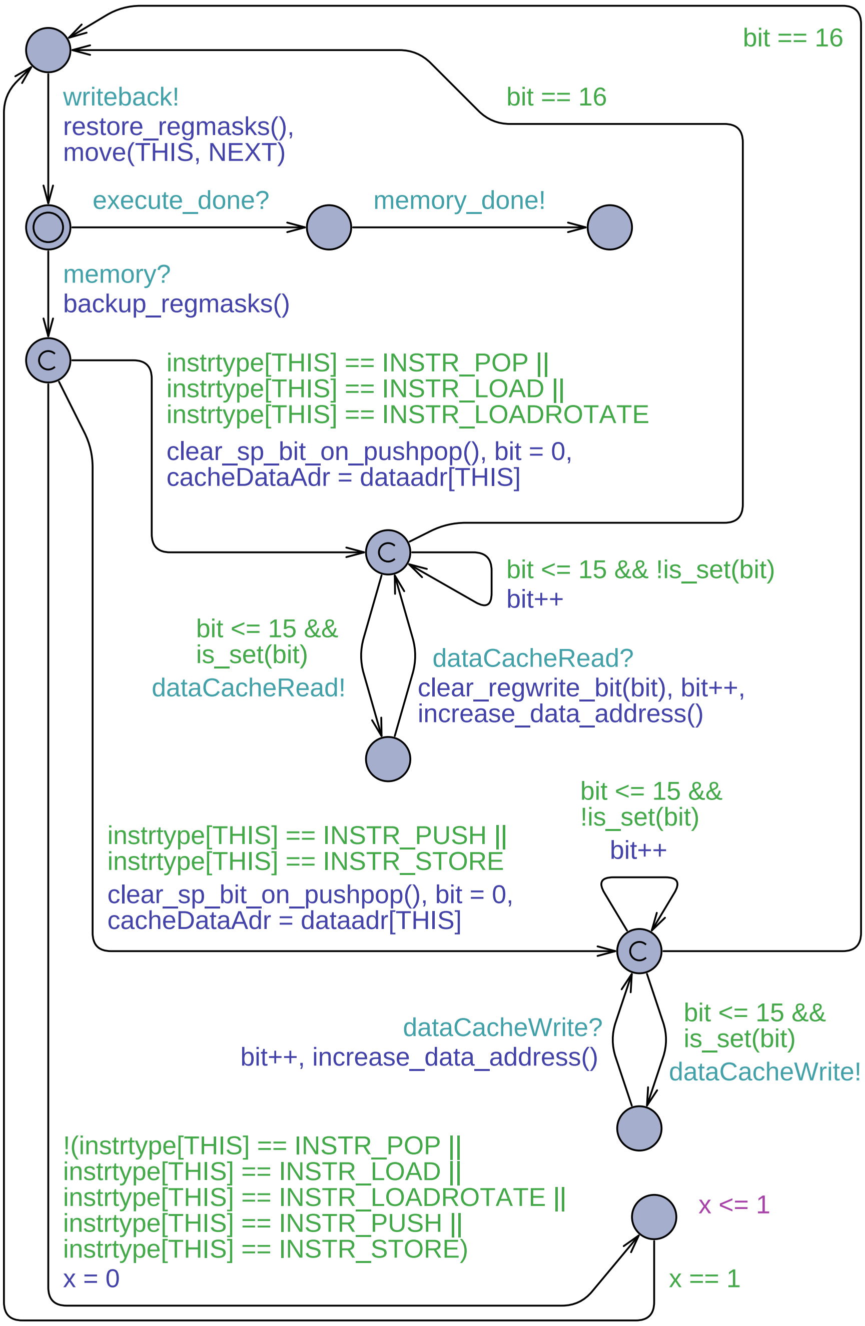

bit == 16
writeback!
restore_regmasks(),
move(THIS, NEXT)
bit == 16
execute_done?
memory_done!
memory?
backup_regmasks()
instrtype[THIS] == INSTR_POP ||
instrtype[THIS] == INSTR_LOAD ||
instrtype[THIS] == INSTR_LOADROTATE
clear_sp_bit_on_pushpop(), bit = 0,
cacheDataAdr = dataadr[THIS]
bit <= 15 && !is_set(bit)
bit++
bit <= 15 &&
is_set(bit)
dataCacheRead!
dataCacheRead?
clear_regwrite_bit(bit), bit++,
increase_data_address()
bit <= 15 &&
!is_set(bit)
bit++
instrtype[THIS] == INSTR_PUSH ||
instrtype[THIS] == INSTR_STORE
clear_sp_bit_on_pushpop(), bit = 0,
cacheDataAdr = dataadr[THIS]
dataCacheWrite?
bit++, increase_data_address()
bit <= 15 &&
is_set(bit)
dataCacheWrite!
!(instrtype[THIS] == INSTR_POP ||
instrtype[THIS] == INSTR_LOAD ||
instrtype[THIS] == INSTR_LOADROTATE ||
instrtype[THIS] == INSTR_PUSH ||
instrtype[THIS] == INSTR_STORE)
x = 0
x <= 1
x == 1

Figure 5.13: UPPAAL automaton for the pipeline's memory stage.

```
bool is_set(int bit)
{
    register_mask regmask = 1 << bit;

    if (instrtype[THIS] == INSTR_POP ||
        instrtype[THIS] == INSTR_LOAD)
        return ((regwrite[THIS] & regmask) > 0); // Load
    else
        return ((regread[THIS] & regmask) > 0); // Store
}
```

Since it is possible that the memory address accessed by a load or store instruction is set to `INVALID_ADDRESS` by the value analysis, the address must only be increased between every load or store if it is different from this special value. The function `increase_data_address` ensures that behaviour:

```
void increase_data_address()
{
    if (cacheDataAdr != INVALID_ADDRESS)
        cacheDataAdr += 4;
}
```

The effect of the `increase_data_address` function could easily be implemented using an extra transition in the memory automaton, but the applied approach has the advantage that it does not increase the state space.

To make a stalling decode automaton move on, bits in the register mask `regwrite[THIS]` must be cleared for every issued load operation when handling a load instruction. The function `clear_regwrite_bit` takes care of this:

```
void clear_regwrite_bit(int bit)
{
    regwrite[THIS] &= bitwise_neg(1 << bit);
}
```

In case of `POP` and `PUSH` instructions, the `SP` (stack pointer) register is always set in both `regread[THIS]` and `regwrite[THIS]`, since these instructions manipulate the stack. When executing a `POP` or `PUSH` instruction in hardware, the `SP` register is used as a start address and is continuously updated such that it always points to the top of the stack, however, it is not one of the registers being loaded or stored and thus should not trigger a load or store operation. The function `clear_sp_bit_on_pushpop` ensures this by clearing the `SP` bit if the instruction is a `POP` or `PUSH`:

```
void clear_sp_bit_on_pushpop()
{
    if (instrtype[THIS] == INSTR_POP)
        regwrite[THIS] &= bitwise_neg(REG_SP);
    if (instrtype[THIS] == INSTR_PUSH)
        regread[THIS] &= bitwise_neg(REG_SP);
}
```

The `backup_regmasks` and `restore_regmasks` functions, called by the memory automaton before and after handling an instruction, back up and restore the register masks, respectively. The masks are changed by the functions

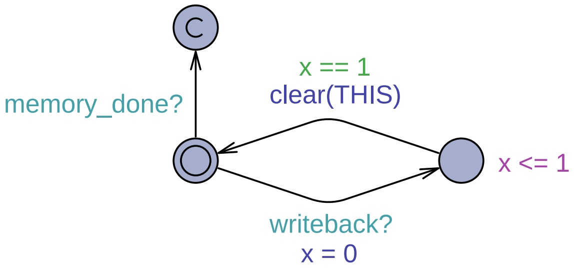


Figure 5.14: UPPAAL automaton for the pipeline's writeback stage.

`clear_sp_bit_on_pushpop` and `clear_regwrite_bit`, and the `regmasks` functions enable the memory automaton to forward unaltered masks to the writeback automaton.

Figure 5.14 depicts the automaton for the writeback stage. Regardless of the instruction type, the automaton delays for one cycle and then clears its instruction data using the function `clear`. Clearing of the instruction data is necessary in order to make the decode automaton move on in case it is stalling due to a dependency on the instruction in the writeback stage.

5.4 Path Analysis

The objective of the path analysis is to find a path through the process' control flow that leads to the WCET. The path analysis presented here is data-insensitive. The analysis is done by reconstructing the CFG of the process and modelling it as a NTA. The automata simulate an abstract execution of the instructions of the process by interacting with the pipeline automata. The CFG is reconstructed by the ARM-to-UPPAAL compiler, which is given the process in an executable form. The compiler uses Dissy, a graphical frontend to the objdump disassembler, to recover information about branches. Branches can either be unconditional or conditional and are either forward or backward jumps. Furthermore, branches can be static or dynamic, where only the former type of branches are currently supported by the ARM-to-UPPAAL compiler. Another flow construct is to write directly to the `PC` (program counter) register, with a possibly dynamically computed value, which makes the control flow hard to determine. In this case the ARM-to-UPPAAL compiler, currently, returns an error message. The construct could be handled by using non-determinism to jump to each of the possible address values the `PC` register can assume, as determined by the value analysis. However, our value analysis returns unknown if more than one value is possible, meaning the non-determinism would have to be that any instruction in the program can be the next, leading to enormous state space explosion. With a better value analysis, this could be handled more elegantly, but currently even a single unknown value would give rise to problems.

Static branches might also result in non-determinism. For instance, a conditional branch gives rise to non-determinism. To decrease the amount of non-determinism, the ARM-to-UPPAAL compiler attempts to force conditional branches to only consider the path leading to the highest WCET. An example of this can be seen in Figure 5.15.

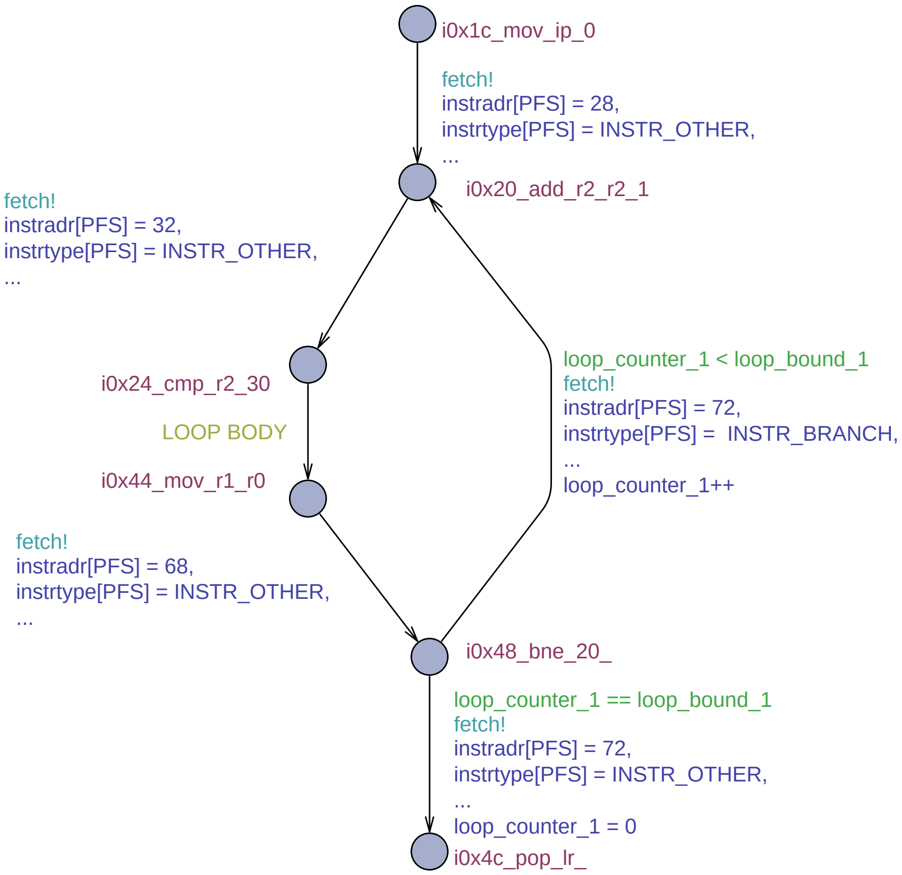


Figure 5.15: A conditional branch is forced to only take the path leading to the highest WCET. PFS is an abbreviation for PIPELINE_FETCH_STAGE.

Figure 5.15 shows the result generated by the compiler on the assembly code in Figure 5.16 on the next page. The compiler reconstructs the control flow, and in this context an important part of the reconstruction is to detect loops to model the behaviour of the process correctly. As data is abstracted away, loop bounds are used to avoid infinite loops. The loop bounds are annotated in the source code, and the executable is compiled with debug information which can identify relevant lines in the source code. The loop bounds are retrieved by the compiler from the source code using the debug information in the executable. This approach can potentially be unsafe in cases where the compiler unrolls a loop.

As can be seen in Figure 5.15, it is only possible to take the sequential transition from location i0x48_bne_20_ and leave the loop, when loop_counter_1 is equal to the loop bound. The reader might notice that instradr[PFS] = 72 on both outgoing transitions of location i0x48_bne_20_. These are added to ensure that the execution of the branch instruction is simulated regardless of

```
Address   Instruction

...
0x0020    ADD R2, R2, #1
0x0024    CMP R2, #30
...
LOOP BODY
...
0x0044    MOV R1, R0
0x0048    BNE 0x20
...
```

Figure 5.16: Assembly code for a loop.

which transition is taken. It would be unsafe if the branch instruction was not simulated on the sequential transition. This gives an idea of how handling jumps in an efficient and correct manner can be complicated. The ARM-to-UPPAAL compiler does not handle arbitrary jumps and produce unsafe WCETs if the executable of the process contains nested loops where the inner loop has a jump to the outer loop. Control flow analysis could be added to the method in order to handle this.

The total determinism in loops, as shown in Figure 5.15, is only possible as long as the hardware platform does not exhibit timing anomalies. Otherwise, it might be unsafe to eliminate the non-determinism. A safe way to eliminate non-determinism is to make the path analysis data-sensitive. This could be done by extracting data flow information from the process and annotate the generated NTA.

The modelling of the CFG as a NTA is done by modelling each function as a timed automaton and modelling function calls as synchronisation between them. In functions, almost every transition simulates an abstract execution of an instruction. The abstract simulation of an instruction is done by synchronising with the timed automaton modelling the pipeline's fetch stage. Organising functions in this way disallows jumps from the body of one function to the body of another function. This is, however, considered bad practice and an error will be produced by the ARM-to-UPPAAL compiler if encountered.

In the following, the transitions that do not simulate an abstract execution of an instruction will be described. These transitions can be divided into: the transition that handles initialisation, transitions that synchronise to realise function calls, and the last transition in a function. The transition used in the process' `main` function to initialise the entire NTA is shown in Figure 5.17 on the facing page. When the transition is taken, the urgent broadcast channel `initCaches` is synchronised over, which will initialise the cache models. Furthermore, the `initialise` function is called which initialises the pipeline stages.

As mentioned above, function calls in the assembly code are found and then modelled as synchronisation between timed automata. An example of a function call is shown in Figure 5.18 on the next page. The figure has three transitions: the first going from location `i0x60_bl_0_` to location `call_fib_0`, the second from location `call_fib_0` to location `return_fib_0`, and an outgoing transi-

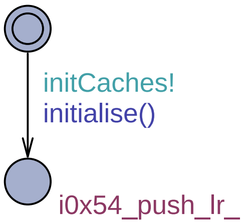


Figure 5.17: Initialisation done by the timed automaton modelling the process' main function.

tion from `return_fib_0`. The first transition simulates the branch instruction entering the pipeline. To pass arguments to the function, the arguments should be pushed on the stack or put into registers. This is already done by the instructions leading up to the branch instruction. The second transition synchronises with the `fib` function using the urgent channel `fib_branch`.

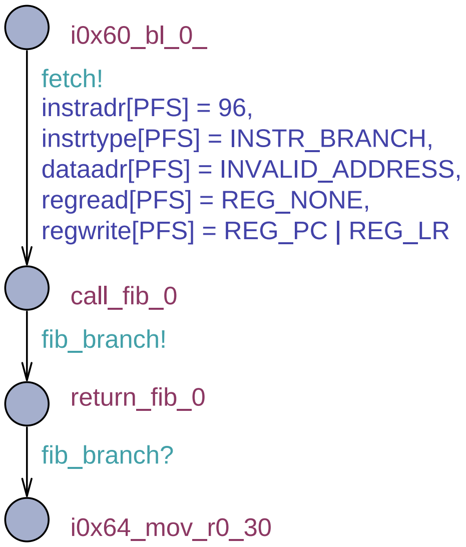


Figure 5.18: Example of a function call.

The relevent part of the `fib` function is shown in Figure 5.19. The synchronisation transfers control to the `i0x0_cmp_r0_1` location of the `fib` function. Besides transfering control, the synchronisation resets the loop counter `loop_counter_1`. The outgoing transition of location `i0x0_cmp_r0_1` starts the abstract simulation of the instructions in `fib`. The remaining function body, connecting location `i0x4_push_lr_` and `i0x50_bx_lr` has been removed for brevity. The ingoing transition to the inital location in Figure 5.19 on the following page transfers control back to location `i0x64_mov_r0_30`. There might be more than one ingoing transition to the initial location in a function, as it might be possible to return from the function in several places. Currently, it is assumed that every `bx lr` instruction in a function is a return. This assumption might give rise to problems in handwritten assembly, since this does not necessarily follow the patterns used by GCC. Furthermore, the current implementation of function calls does not support assembly-level recursion, which is in fact unsafe in the current implementation. This limitation is not really a problem, as recursion is normally not used in RTSs.

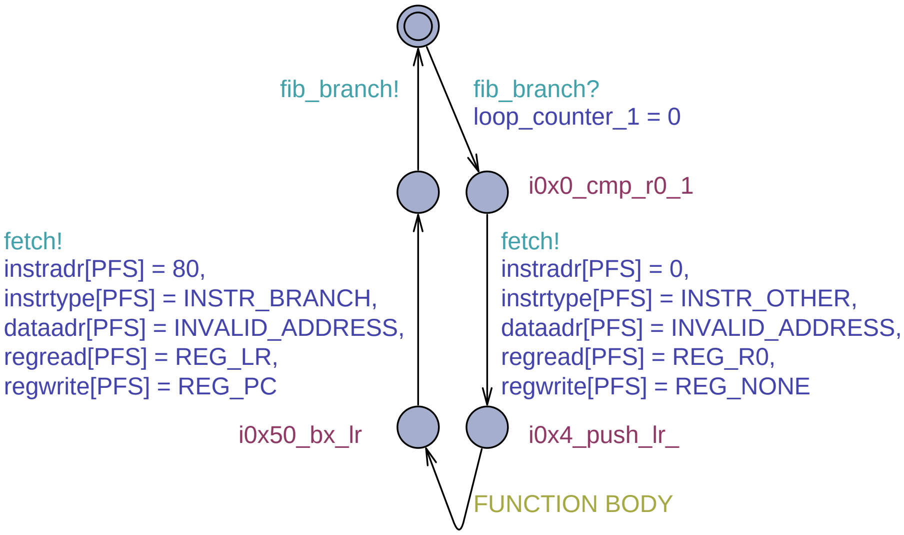


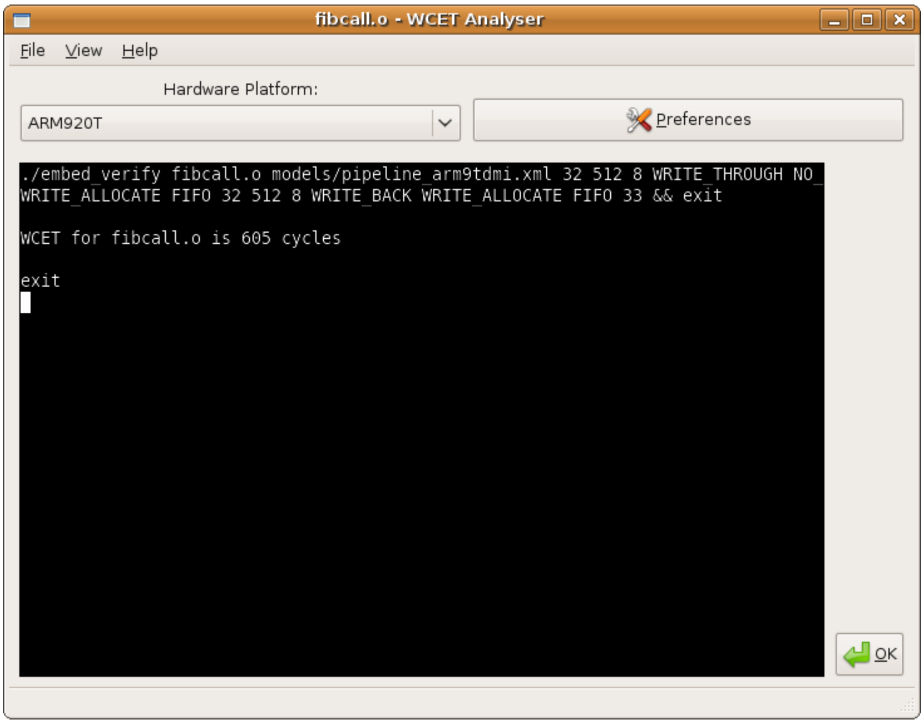
Figure 5.19: Example of the beginning of a function automaton.

5.5 METAMOC Graphical User Interface

A graphical user interface (GUI), which connects the different implementation components, has been developed to make the implementation accessible for end-users. The main window of the tool is shown in Figure 5.20 on the next page. The figure shows the main window after the object file `fibcall.o`, a benchmark from the WCET Challenge 2006 [4], has been analysed. After a click on the "OK" button, the WCET Analyser displays the WCET of the selected object file, which in this case is 605 cycles. Since the WCET depends on the hardware platform, the GUI enables the user to select different platforms for the analysis. In the demonstrated setting, the chosen platform is the ARM920T.

By clicking the "Preferences" button in the top of the interface, it is possible to make modifications to the platform. The click opens the window displayed in Figure 5.21 on the facing page. The window allows the user to tryout variations of the platform, e.g. it is possible to vary the delay induced by the main memory. It is also possible to try out various settings for caches. This is done by clicking the icons next to the caches.

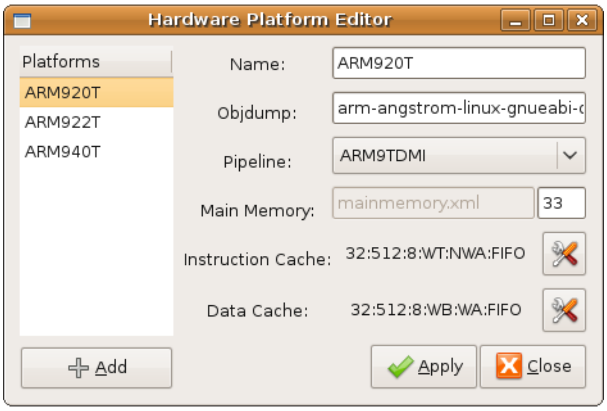
Clicking these buttons brings up a window as displayed in Figure 5.22. In this window it is possible to change the replacement policy of the cache and the other settings listed in the window.



Hardware Platform: ARM920T

```
./embed_verify fibcall.o models/pipeline_arm9tdmi.xml 32 512 8 WRITE_THROUGH NO_
WRITE_ALLOCATE FIFO 32 512 8 WRITE_BACK WRITE_ALLOCATE FIFO 33 && exit

WCET for fibcall.o is 605 cycles

exit
```

Figure 5.20: The main window of WCET Analyser.



Name: ARM920T

Objdump: arm-angstrom-linux-gnueabi-c

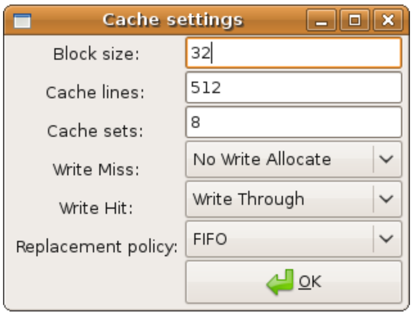
Pipeline: ARM9TDMI

Main Memory: mainmemory.xml 33

Instruction Cache: 32:512:8:WT:NWA:FIFO

Data Cache: 32:512:8:WB:WA:FIFO

Figure 5.21: The hardware platform editor window of WCET Analyser.



Block size: 32

Cache lines: 512

Cache sets: 8

Write Miss: No Write Allocate

Write Hit: Write Through

Replacement policy: FIFO

Figure 5.22: The cache settings window of WCET Analyser.

Chapter 6

Experiments

In this chapter we document a series of experiments conducted on the concrete implementation that served as an ongoing example in Chapter 5. The purpose of the experiments is to derive properties of the METAMOC method, which will reveal how well the method works in practice and disclose weak points that can serve as basis for future development. Three qualities are examined: how big and how complex programs the method is able to analyse, how much sharper WCETs the method is able to determine when taking caching into account, and how fast the method is able to analyse processes.

The experiments are conducted using a number of WCET analysis benchmark programs made available in the WCET Analysis Project by Mälardalen Real-Time Research Center (MRTC) [4]. The programs implement a wide selection of tasks that are typical for embedded software systems, such as signal filtering, data compression, sorting, matrix calculation, and numerical calculation. In addition, the programs let us test the method's strength against different types of control flow, as they cover loops, nested loops, recursion and even unstructured code.

Unfortunately, no reference WCETs for running the benchmark programs on an ARM920T processor are available. These numbers would have been useful for assessing whether the WCETs determined by our implementation are safe and to which degree they are sharp. Other researchers and research companies in the field of WCET analysis, such as Daniel Sandell [37], who has had his results published as an advertisement for the AbsInt aiT tool [5], use WCETs from the emulator ARMulator by ARM Ltd. This approach does not give rise to sound comparisons, as ARM Ltd. does not give any guarantees regarding the cycle-accuracy of ARMulator [26, Section 2.1.1]. Because not all cycle relevant information about ARM processors is deducible from the official, technical reference manuals, and because measurements of execution time cannot be trusted, the only options for obtaining usable numbers might be to persuade ARM Ltd. into releasing detailed hardware descriptions or to reverse engineer the hardware.

Our experiments have been conducted in the following manner: We first manually annotate loop bounds in the C programs that are not already annotated by their original authors. This is done using overestimates in some cases, because some bounds are difficult to determine, and we want to ensure that the bounds do not give rise to underapproximations. Each benchmark program is compiled using a version of GCC from one of the OpenMoko cross-compilation toolchains [2] at three different optimisation levels: `-O2`, `-O1` and `-O0`. For all levels we have used the additional switches `-g`, for including debugging information, and `-fno-builtin`, for only using functions directly defined in the source code. The resulting binaries are then fed into our implementation, and the total time from invocation to the time a result is presented to the user is measured. The UPPAAL model checking is done on a different host than the rest of the analysis, so a slight, negligible, network overhead for transferring the resulting UPPAAL model to another host is included in the timings. The primary host is a Lenovo SL500 laptop with a 2 GHz Intel Core 2 Duo processor and 4 GB of RAM running a 32 bit Ubuntu operating system, while the model checking host is a Dell PowerEdge 2950 with two 2.5 GHz Intel Quad Core Xeon processors and 32 GB of RAM running a 64 bit Red Hat operating system. The model checking is run on the more powerful system in order for UPPAAL to be able to utilise the entire 4 GB of RAM that UPPAAL is able to use.

We have tried seven different combinations of the optimisation levels together with three different settings for the caches and the value analysis:

- Instruction cache is always miss, data cache is always miss.
- Instruction cache is modelled concretely, data cache is always miss.
- Instruction cache is modelled concretely, data cache is modelled concretely, and the value analysis is used.

The results of our experiments are presented in Tables 6.1 and 6.2, starting at page 84. The three most interesting findings are:

Taking the instruction cache into account yields WCETs that are up to 97% smaller. For example, the `janne_complex` example goes from a WCET of 8908387 to 270083. The average decrease in WCET is 78% at the optimisation level `-O2`.

Using a concrete data cache gives up to 68% smaller WCETs. At the optimisation level `-O2`, the average decrease in WCET is 31%. The gain from taking the data cache into account is very dependent on the value analysis being able to determine values for memory accesses. The `cnt` program goes from a WCET of 52481 to 16828 when taking the data cache into account. For `cnt`, the value analysis is able to determine values for about half of all memory accesses[1].

Almost all results are obtained within five minutes. Only four of the results are obtained after five minutes, and the longest analysis time is less than 25 minutes. This is disregarding the analyses that run out of memory, as these have not been timed. We have, however, observed timings of more than four hours, before UPPAAL ran out of memory.

The remaining findings, which point out some surprising facts and uncover some limitations of the current state of the implementation, are:

[1] The `cnt` program sums up all elements in a two-dimensional array. The value analysis loses the value of the pointer to the array, as it is stored on the stack and later retrieved, between the initialisation of the array and the summation of the array.

The most optimised level is not necessarily the one resulting in the smallest WCET, among the analysis results. Looking at the `jfdctint` program, the smallest WCET, 32043, is found at the optimisation level `-O1` and, unsurprisingly, using a concrete instruction and data cache. The WCET is 53% smaller than the WCET found with `-O2`, 68740. The example program `expint` is unanalysable at `-O1` but analysable at `-O0`.

Seven programs cannot be analysed, as the model checking runs out of memory due to state space explosion. The most common cause of state space explosion, that we have identified, is related to the path analysis failing to recover enough flow information. Since we only unroll loops in the value analysis — not in the path analysis — a typical example is a loop with a conditional forward branch for escaping the loop. In this case the path analysis uses non-determinism at each loop iteration, leading to an exponential blow-up.

Two programs write to the program counter, leading to a dynamic control flow. One of the cases is a switch statement, which is compiled into a lookup table, the other is the well-known, not completely well-structured program "Duff's device".

Six programs use floating point operations, and two programs use floating point operations at lower optimisation levels. Our implementation currently does not handle floating point operations, as these are implemented as library calls in the ARM architecture. It would be simple to support these, provided that good loop bounds could be deduced from the very optimised assembler code for these procedures.

The value analysis is able to analyse all but four examples. The implementation fails with a Python exception, but we have not looked into why.

	-O2, **miss DC,** **miss IC,** **no value** **analysis**	-O2, **miss DC,** **concrete IC,** **no value** **analysis**	-O2, **concrete DC,** **concrete IC,** **value analysis**
adpcm	out of memory		
bs	3666 00:01.06	964 00:01.19	640 00:01.36
bsort100	out of memory		
cnt	188466 00:01.99	52481 00:02.63	16828 00:04.34
compress	out of memory		
cover	out of memory		
crc	2043591 00:46.84	328310 01:15.65	value analysis fails
duff	write to PC		
edn	2093178 00:14.46	686447 00:22.03	out of memory
expint	658716 00:04.65	26417 00:06.21	value analysis fails
fac	11553 00:01.15	1266 00:01.25	907 00:01.54
fdct	out of memory		
fft1	uses floating point		
fibcall	13731 00:01.01	734 00:01.05	605 00:01.32
fir	131508 01:00.26	20824 13:53.00	out of memory
insertsort	75078 00:01.15	43888 00:01.45	36275 00:02.76
janne_complex	8908387 00:38.72	270083 01:09.74	out of memory
jfdctint	294957 00:05.32	133059 00:06.17	68740 05:15.00
lcdnum	write to PC		
lms	uses floating point		
ludcmp	out of memory		

Table 6.1: WCETs and analysis times for the benchmark programs from Mälardalen WCET Research Group compiled with -O2. DC and IC are abbreviations for “data cache” and “instruction cache”, respectively. The time format is “mm:ss.jj”, where mm is minutes, ss is seconds, and jj is centiseconds.

	`-O2`, **miss DC, miss IC, no value analysis**	`-O2`, **miss DC, concrete IC, no value analysis**	`-O2`, **concrete DC, concrete IC, value analysis**
matmult	5836449 00:22.75	2636715 00:38.31	out of memory
minver	uses floating point		
ndes	out of memory		
ns	279579 00:04.28	29060 00:07.41	28572 00:19.41
nsichneu	out of memory		
prime	33231 00:02.48	4532 00:03.41	value analysis fails
qsort-exam	uses floating point		
qurt	uses floating point		
select	uses floating point		
statemate	uses floating point		
ud	out of memory		

Table 6.1 continued.

	`-O1`, **miss DC, miss IC, no value analysis**	`-O1`, **miss DC, conc. IC, no value analysis**	`-O1`, **conc. DC, conc. IC, value analysis**	`-O0`, **miss DC, miss IC, no value analysis**
adpcm	out of memory			
bs	3864 00:01.15	906 00:01.18	679 00:01.38	8979 00:01.47
bsort100	out of memory			
cnt	174474 00:03.96	47810 00:05.77	32597 00:08.61	uses floating point
compress	out of memory			
cover	out of memory			
crc	2043670 00:45.04	329322 01:16.49	out of memory	out of memory
duff	write to PC			
edn	1822726 03:39.94	out of memory	out of memory	8501103 00:47.68

Table 6.2: WCETs and analysis times for the benchmark programs from Mälardalen WCET Research Group compiled with `-O1` and `-O0`. DC and IC are abbreviations for “data cache”, “instruction cache”, respectively. The time format is “mm:ss.jj”, where mm is minutes, ss is seconds, and jj is centiseconds.

	-O1, **miss DC, miss IC, no value analysis**	-O1, **miss DC, conc. IC, no value analysis**	-O1, **conc. DC, conc. IC, value analysis**	-O0 **miss DC, miss IC, no value analysis**
expint	out of memory	out of memory	out of memory	2661123 24:47.44
fac	18483 00:11.21	6323 00:18.78	out of memory	30627 00:13.35
fdct	out of memory			
fft1	uses floating point			
fibcall	14424 00:01.01	661 00:01.18	628 00:01.31	44520 00:01.27
fir	144048 01:04.22	20932 05:04.35	out of memory	342180 00:35.01
insertsort	74946 00:01.10	43878 00:01.43	40439 00:02.39	175366 00:01.64
janne_complex	out of memory			
jfdctint	150978 00:02.90	36159 00:03.63	32043 00:35.53	370923 00:04.62
lcdnum	write to PC			
lms	uses floating point			
ludcmp	uses floating point			
matmult	out of memory			
minver	uses floating point			
ndes	out of memory			
ns	12741 00:01.75	1885 00:02.46	1655 00:03.65	482463 00:06.19
nsichneu	out of memory			
prime	32904 00:02.38	4612 00:03.51	value analysis fails	out of memory
qsort-exam	uses floating point			
qurt	uses floating point			
select	uses floating point			
statemate	uses floating point			
ud	out of memory			

Table 6.2 continued.

Chapter 7

Related Work

A large amount of literature exists which treats WCET analysis. As mentioned in Section 2.2 on page 14 there are two classes of methods for finding WCETs: measurement-based methods and static methods. Several tools exist, which are based on measurement-based methods. Such tools are, however, not suitable for hard real-time systems, as they do not guarantee safe WCETs, and we therefore do not look further into measurement-based methods. For hard real-time systems, tools based on static methods should be used. In [6, 19, 20, 23, 46, 45], the authors present a number of complementary static methods used in the state-of-the-art, static WCET analysis tool aiT. The underlying techniques used in aiT are abstract interpretation and implicit path enumeration technique (IPET).

Abstract interpretation is a well-known technique from the area of static program analysis that allows for formal correctness proofs of analyses. To ease the development of analysers, the tool PAG [32] has been developed. PAG allows generating an implementation of an analysis based on a specification of the analysis. The aiT tool uses this to develop analysers. IPET is used to find the longest execution path of a process. It requires safe execution times of basic blocks together with execution counts of the basic blocks to find a WCET. The IPET calculation often uses integer linear programming (ILP) by making an ILP formulation of the problem. The ILP consists of two elements: a cost function and constraints on the variables in the cost function. The cost function represents the number of cycles, and to find the WCET of a process, the cost function is maximised. The constraints on the variables represent the number of times the basic blocks can be executed.

A WCET analysis is often separated in, more or less, the following four analyses: cache, value, pipeline and path analysis. In the context of cache analyses they can generally be sorted into abstract and concrete cache analyses. The abstract cache analyses are based on similar methods as described in Section 3.4.1 on page 36 and have been implemented in [19] using PAG and a NTA with stop watches [7]. Abstract caches have the advantage of being space efficient. The trade-off is a loss in the precision of the WCET results. With concrete caches as implemented in [33], more precise results can be achieved. An advantage of a more precise result of the cache analysis might also decrease the amount of non-determinism for processors with timing anomalies. It should be noted that both concrete and abstract cache analyses must handle unknown memory addresses, since the value analysis might not be able to determine all adresses.

The pipeline analysis typically uses an abstract model of the pipeline to take its impact on the execution into account [33, 19, 20, 23]. There are, however, exceptions such as the approach presented in [17]. In [17], an unmodified processor simulator is used together with IPET. The approach has a number of limitations, e.g. the simulator must be cycle-accurate and it must be possible to control the state of the simulator. Furthermore, the processor must not have timing anomalies [44]. The pipeline analysis should, like the cache analysis, be able to handle unknown memory values. Unknown memory values might lead to non-determinism, as it might be impossible to make a reasonable overapproximation. For this reason, abstract pipeline states are traditionally represented as a set of concrete pipeline states [38]. Recent work has looked into using binary decision diagrams (BDDs) to represent abstract pipeline states combined with model checking [46, 45]. The work presented in this thesis is conceptually similar, although instead of using BDDs, the standard techniques implemented in UPPAAL are used for state space reduction. Another interesting perspective is the work presented in [18], where the authors try to bound the distance of the effect of instructions. That is, they try to determine a bound on the number of subsequent instructions following an instruction, that the instruction can affect the timing of. This is only initial work in this area. They do, however, show that many pipelines allow arbitrary long sequences to be affected by an instruction, leaving little or no hope for the approach.

Value analysis is used to determine addresses of memory blocks, which are accessed dynamically using registers and variables [44, p. 12]. Furthermore, it is used to find loop bounds and thereby also to find infeasible paths. A type of value analysis is value set analysis, which finds supersets of possible values. To get improved results, value set analysis should be used in combination with an affine-relations analysis. In [9], a value set analysis with affine relations for x86 code has been implemented. In the aiT tool, the value analysis has been implemented using PAG. The value analysis in this thesis is currently based on constant propagation using WPDSs [36]. WPDSs have the advantage, compared to PAG, of supporting a broader range of data flow analysis queries [36, p. 4]. We do, however, not exploit this in the current implementation.

For the path analysis, IPET and ILP have been combined in several tools [44, p. 42]. In [41], a so-called path-based method is presented and has been implemented as an alternative to IPET in the SWEET tool. The method is more effective than previous path based methods. Comparing to IPET, it should be noted that finding the solution to an ILP problem is NP-hard. Furthermore, path based methods explore the path explicitly. In contrast to IPET, this could make debugging of models easier. The path analysis presented in this thesis is simple exploration of the CFG of the process. Not much optimisation has been done, and this is a limiting factor of the implemented METAMOC method.

In [20] it is claimed that it is impossible to make the four analyses as separate, modular analyses and at the same time get sufficient precision. They define a separate and modular analysis to be an analysis, where sub-analyses can be run independently or as a sequence of subtasks. The paper is based on experiences from enabling the aiT tool to find WCETs for the two rather advanced processors: ColdFire 5307 and PowerPC 755. Our analysis is not separate in the same sense as in [20], but it is modular to a great extent nonetheless.

It also seems that the statement in [20] does not cover the value analysis.

In the aiT tool, the value analysis is executed by itself and the output is given as input to the other analyses. The advantage of separating the value analysis from the rest of the analyses is that the other analyses can be data-insensitive and thus require less memory compared to the method in [30], where a symbolic simulation, that can handle unknown input using non-determinism, is used.

The method presented in this thesis separates the four analyses by running the value analysis prior to the other analyses, like the aiT tool. The cache, pipeline and path analyses are then carried out by separate models that communicate. Unlike the tools Bound-T, aiT and SWEET, all three analyses are performed simultaneously. Even though the models are separate, time elapses globally. To reduce the amount of memory used by state space exploration, we use a number of modelling tricks: the state space is explored in depth-first order, the models have as little non-determinism as possible, and the stored state space is reduced as much as possible. Currently, the caches are modelled concretely. We have, however, experimented with implementing an abstract cache analysis, as in [6], using real-time model checking, which is covered in [7].

Chapter 8

Conclusion

Modern processors have a number of features for increasing the average case performance, which unfortunately make it hard to determine a safe and sharp WCET. Among the possible features of modern processors, especially caching and pipelining are central to performance improvement, and it is therefore vital to take them into account. In this thesis we have developed the modular WCET analysis method METAMOC, which accomplishes exactly that, and we have furthermore conducted a series of experiments on an implementation of the method. The implementation is for the ARM920T processor, which is a typical processor for embedded systems. The experiments suggest that WCET analysis methods based on model checking can be a successful solution to the problem of finding safe and sharp WCETs for processes running on processors featuring caching and pipelining. Firm baseline numbers are, however, still required in order to obtain solid evidence.

The method is composed of the four common WCET sub-analyses and the interaction between them. The sub-analyses are cache, value, pipeline and path analysis. We have designed a modular, general and — to some degree — scalable method for performing WCET analysis. The method is modular in that the sub-analyses are done in an integrated but loosely coupled manner. Besides being the basis for experiments, the implementation of the method serves as a successful demonstration.

The experiments show a noticeable improvement in WCETs caused by the instruction and data cache analyses. The concrete instruction cache analysis gives an average of 78% smaller WCETs, and the concrete data cache analysis gives an average of 31% smaller WCETs. The decrease in WCETs caused by the data cache analysis depends on the result of the value analysis. Since the value analysis is relatively simple, a more advanced value analysis, which determines register contents in more execution points, is expected to further decrease the WCETs. Taking into consideration that the value analysis is relatively simple, the analysis is still able to produce results that give rise to noticeable smaller WCETs, and it seems to be a good starting point for improvements of the implementation.

Additionally, the experiments show that all the benchmark programs, which the method are able to analyse, can be analysed within five minutes. There are, however, limiting factors. For example, floating point operations are currently not handled, which makes the method fail on six programs. Furthermore, seven of the programs makes the model checking part of the analysis run out of memory. This seems to be caused by not preventing non-determinism of conditional forward branches, which lead to state space explosion. To overcome these limitations, an improved path and value analysis should be designed and implemented.

Chapter 9

Future Work

The work done in this thesis gives rise to a number directions for future work. The overall goal of the future work is to improve the developed METAMOC method and associated implementation, make the method practically usable by combining it with scheduling of RTSs in a ready-to-use, user-friendly tool, and study whether the ideas behind the method can be applied to software verification in general. In the following paragraphs we introduce some of the directions, many of which have enough substance to give rise to one or more papers in the domain of software verification.

Firstly, the concrete cache models limit the scalability of the developed method, as they increase the size of the state space considerably. An improvement would be to invent space-saving, abstract cache models, which we have already studied to some degree in [7]. The challenge is to devise an abstraction that utilises the time aspect of timed automata and thus takes advantage of existing optimisation techniques for verifying properties of this type of automata. The studied approach is, however, not modular. New, specialised data types in the UPPAAL model checker might help solve this problem. Another strategy could be to decrease the size of the concrete caches. Each cell in the instruction cache only stores a limited number of the combinations possible with 32 bit, implying that the number of bits used for each cell could be decreased. As we have noticed that most of the WCET benchmark programs do not fill the entire data cache, the number of cells in each set in the cache could be decreased.

Compared to the other sub-analyses utilised by the METAMOC method developed in this thesis, not much consideration has been given to the path analysis. Since the method takes a binary executable as input in order to capture the effects of compiler optimisations, the result of path analysis depends on how well the control flow can be reconstructed. Moreover, the addition of safe, limiting guards, which eliminate infeasible and unnecessary paths from the CFGs, will improve the scalability of the path analysis and thus of the entire WCET analysis. An interesting approach is path-based path analysis, as presented in [41]. The applied value analysis is another sub-analysis that can be improved by integrating more sophisticated methods. For example, value analysis methods already developed, such as the value set analysis in [9], could be desirable candidates.

The way hardware components and process functions currently are modelled places some restrictions on the possible control flow. For example, recursion is not supported in a scalable way, and jumps from one function to arbitrary places in another function are impossible. Even though the latter example is considered bad practice, all sorts of jumps are possible at the assembler code level. It is possible that arbitrary static jumps, similar to the one mentioned, are inserted by optimising compilers, which makes it necessary for WCET analyses to handle them. By representing the instruction sequence of a process as a constant array in UPPAAL and adding a driver automaton that acts as an instruction pointer and manages a call stack, it would be possible to handle all types of jumps.

Another direction for future research is to combine the work done in this thesis with the work done by Bøgholm et al., in connection with the tool *Schedulability Analyzer for Real-Time Systems (SARTS)* [12]. In SARTS, automata for the processes in a RTS are combined with an automaton for a scheduler to form a NTA, for which UPPAAL is used to verify the schedulability of the RTS. Inspired by SARTS, the method presented in this thesis could be further developed to support a real-time scheduler and multiple processes, resulting in a method for schedulability analysis with support for hardware platforms featuring caching and pipelining.

Even though NTAs and UPPAAL work reasonably well for the method at this stage, it might prove rewarding to experiment with other types of models and other model checkers. A nuisance with the current approach, which causes problems that are documented in Chapter 6 starting on page 81, is that UPPAAL — in its current incarnation — lacks support for more than 4 GB of memory. With regard to other types of models, two fellow students at Department of Computer Science, Aalborg University, Joakim Byg and Kenneth Yrke Jørgensen, have created the tool TAPAAL [14], which makes it possible to verify properties of Petri nets using UPPAAL.

A way to overcome UPPAAL's memory limitation, and in general decrease the METAMOC method's memory usage, could be to extend UPPAAL with an aggressive, specialised reachability optimisation technique. The technique should utilise that UPPAAL does not need to store visited states in memory, since the NTAs generated by the method always deadlock at some point, and visited states are therefore never revisited.

In order to evaluate the results of our experiments more thoroughly, the real WCETs for running the benchmark programs on the ARM920T are needed. Establishing the WCETs theoretically using the information published in the manuals from ARM Ltd. is of no value, as this information was also used for the presented implementation, and the results would therefore not differ. Obtaining WCETs using the emulator ARMulator is not enough either, since ARM Ltd. does not guarantee that the emulator is cycle-accurate [26, Section 2.1.1]. The results from ARMulator might be good enough for approximate comparisons, but determining the associated uncertainty will require more detailed information about the emulator's accuracy than currently available. Running the benchmark programs on actual hardware featuring the ARM920T, such as the OpenMoko FreeRunner mobile phone [2], might be useful for finding lower bounds on the WCETs, for comparison.

From the experience gained by developing the presented method, we believe that the behaviour of hardware platforms with timing anomalies can be imitated by adding extra non-determinism to the hardware models. By modelling the hardware's functional units, such as LSUs, IUs and MCIUs (see Section 3.3 on page 34), as individual, timed automata and not limiting the execution order using process priorities or similar, the model checker will try out all possible execution orders and determine the worst-case order. The interesting part is to study whether the added non-determinism results in a state space explosion, and, if that is the case, investigate other ways to model timing anomalies.

Since it is essential that implementations of the presented METAMOC method work correctly, i.e. they imitate the hardware platforms in a safe and accurate way, an important direction for future work is to prove this property. For a particular hardware platform, that requires a detailed hardware specification from the manufacturer and an automated, guaranteed correct method for verifying that the handcrafted hardware models implement the specification correctly. If such detailed specifications are available, however, it might be more rewarding to study how specifications written in hardware description languages, such as VHSIC Hardware Description Language (VHDL), can be automatically translated into models usable for WCET analysis.

Currently, the method uses static analysis to perform value analysis and model checking to perform cache, pipeline and path analysis. It should be investigated whether other boundaries between static analysis and model checking could lead to better WCET analyses in terms of speed, capability and precision. Combining static analysis and model checking is a relatively new approach, and a promising direction for future work could be to apply the experience gained from combining them for WCET analysis to software verification in general.

Support for more ARM processors than the ARM920T can easily be added to the presented implementation, as these processors, to a great extent, share a common instruction set. On the other hand, adding support for a hardware platform which uses another instruction set is a sizeable task, since it requires the ARM-to-UPPAAL compiler to be mostly rewritten. Also, some additions to the Dissy disassembler front-end might be needed. If the implementation's codebase is refactored to become more general with regard to dealing with different hardware platforms and is reorganised to become simpler to understand, the job of adding support for more platforms is made substantially easier. This in turn raises the value of the implementation and makes it likely that other researchers in the field of WCET analysis might contribute with support for more hardware platforms and use the software as a basis for future work.

The method could be extended to also check for software problems such as race conditions and resource depletion. The former extension is of course only relevant if a scheduler and more processes are introduced into the method as mentioned above. These extra checks fit in well, since the method is already doing a full exploration of a process' control flow. Similarly, the method could be used on arbitrary parts of the control flow and provide advanced profiling data. For example, software engineers could study the execution of their code and learn how to best utilise caching and pipelining.

Finally, other parameters than WCET could be determined. Engineers and scientists in the world of embedded systems have always been concerned with limiting the energy consumption of electric equipment. Also, people's growing concerns for the environment, together with the effects of the economy crisis that began in late 2008, make the topic of limiting resource usage interesting. With enough information available from hardware manufacturers, the METAMOC method could be altered to estimate the worst-case energy consumption of executing a particular software process.

Bibliography

[1] Homepage of the Danish Network for Intelligent Embedded Systems (DaNES). `http://www.danes.aau.dk`.

[2] Homepage of the OpenMoko mobile phone. `http://openmoko.org`.

[3] Homepage of the SSV 2009 Doctoral Symposium held during the 4th International Workshop on Systems Software Verification (SSV 09). `http://www.embedded.rwth-aachen.de/ssv09/doku.php?id=doctoral`.

[4] The WCET Analysis Project by Mälardalen Real-Time Research Center (MRTC). `http://www.mrtc.mdh.se/projects/wcet/home.html`.

[5] AbsInt. aiT for ARM7 vs ARMulator. `http://www.absint.com/ait/precision.htm#armulator`.

[6] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1996. Springer-Verlag.

[7] Mads Christian Olesen Andreas Engelbredt Dalsgaard and Martin Toft. Worst-Case Execution Time Analysis for Real-Time Systems. DAT5 report, `http://martintoft.dk/projects/dat5.pdf`, December 2008.

[8] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, first edition, 2008.

[9] Gogul Balakrishnan and Thomas W. Reps. Analyzing Memory Accesses in x86 Executables. In *Proc. Compiler Construction LNCS 2985*, pages 5–23. Springer, 2004.

[10] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A Tutorial on Uppaal. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[11] Christoph Berg. PLRU Cache Domino Effects. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[12] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-Based Schedulability Analysis of Safety Critical Hard Real-Time Java Programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.

[13] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Pearson Education Limited, third edition, 2001.

[14] Joakim Byg and Kenneth Yrke Jørgensen. Homepage for TAPAAL. `http://tapaal.net`.

[15] Andreas Engelbredt Dalsgaard, Mads Christian Olesen, and Martin Toft. WCET Analysis of ARM Processors using Real-Time Model Checking. Extended abstract accepted for the *SSV 2009 Doctoral Symposium* held during the *4th International Workshop on Systems Software Verification (SSV 09)*. `http://martintoft.dk/projects/dsssv09.pdf`.

[16] Jochen Eisinger, Ilia Polian, Bernd Becker, Stephan Thesing, Reinhard Wilhelm, and Alexander Metzner. Automatic Identification of Timing Anomalies for Cycle-Accurate Worst-Case Execution Time Analysis. In *DDECS '06: Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems*, pages 13–18, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Jakob Engblom and Andreas Ermedahl. Pipeline Timing Analysis using a Trace-Driven Simulator. In *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA 99, Cat No PR00306)*, pages 88–95, 1999.

[18] Jakob Engblom and Bengt Jonsson. Processor Pipelines and Their Properties for Static WCET Analysis. In *Proceedings of EMSOFT 02: Second International Conference on Embedded Software, volume 2491 of Lecture Notes in Computer Science*, pages 334–348. Springer-Verlag, 2002.

[19] Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. 1997.

[20] Reinhold Hechmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, 91(7):1038, 2003.

[21] Nicholas Kidd, Akash Lal, and Thomas W. Reps. Weighted Automata Library. `http://www.cs.wisc.edu/wpis/wpds/download.php`.

[22] Peter Knaggs and Stephen Welsh. ARM Assembly Language Programming. `http://www.arm.com/miscPDFs/9658.pdf`, 2004.

[23] Marc Langenbach, Stephan Thesing, and Reinhold Heckmann. *Pipeline Modeling for Timing Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 294–309. Springer Berlin / Heidelberg, 2002.

[24] Nancy G. Leveson and Clark S. Turner. *An Investigation of the Therac-25 Accidents*, volume 26 of *Computer*, pages 18–41. IEEE, July 1993.

[25] ARM Ltd. ARM9 - ARM Processor Family. http://www.arm.com/products/CPUs/families/ARM9Family.html.

[26] ARM Ltd. ARM Developer Suite – Debug Target Guide. http://infocenter.arm.com/help/topic/com.arm.doc.dui0058d/DUI0058.pdf, 1999.

[27] ARM Ltd. ARM9TDMI Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0180a/DDI0180.pdf, 2000.

[28] ARM Ltd. ARM920T Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf, 2001.

[29] ARM Ltd. ARM and Thumb-2 Instruction Set Quick Reference Card. http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf, March 2007.

[30] Thomas Lundqvist. *A WCET Analysis Method for Pipelined Microprocessors with Cache Memories.* PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2002.

[31] Thomas Lundqvist and Per Stenström. Timing Anomalies in Dynamically Scheduled Microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, Washington, DC, USA, 1999. IEEE Computer Society.

[32] Florian Martin. PAG – An Efficient Program Analyzer Generator. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):46, 1998.

[33] Alexander Metzner. Why Model Checking Can Improve WCET Analysis. In *Proceedings of Computer Aided Verification*, pages 334–347. Springer Berlin / Heidelberg, 2004.

[34] David A. Patterson and John L. Hennessy. *Computer Organization and Design.* The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science & Technology, third edition, 2004.

[35] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A Definition and Classification of Timing Anomalies. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[36] Thomas W. Reps, Akash Lal, and Nick Kidd. *Program Analysis using Weighted Push-Down Systems*, volume 4855 of *Lecture Notes in Computer Science*, pages 23–51. Springer Berlin / Heidelberg, 2007.

[37] Daniel Sandell. Evaluating Static Worst-Case Execution-Time Analysis for a Commercial Real-Time Operating System. Master's thesis, Department of Computer Science, Mälardalen University, Sweden, July 2004. http://www.mrtc.mdh.se/publications/0738.pdf.

[38] Jörn Schneider and Christian Ferdinand. Pipeline Behavior Prediction for Superscalar Processors by Abstract Interpretation. In *LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems*, pages 35–44, New York, NY, USA, 1999. ACM.

[39] Stefan Schwoon. *Model-Checking Push-Down Systems.* PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, 2002.

[40] Simon Segars. The ARM9 Family – High Performance Microprocessors for Embedded Applications. In *ICCD '98: Proceedings of the International Conference on Computer Design*, pages 230–235, Austin, TX, USA, 1998. IEEE Computer Society.

[41] Friedhelm Stappert, Andreas Ermedahl, and Jakob Engblom. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 01)*, pages 132–140, New York, NY, USA, 2001. ACM.

[42] Andrew S. Tanenbaum. *Structured Computer Organization.* Pearson Education, fourth edition, 1998.

[43] Stephan Thesing. *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models.* PhD thesis, Department of Computer Science, Saarland University, Germany, July 2004.

[44] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem — Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[45] Stephan Wilhelm. Efficient Analysis of Pipeline Models for WCET Computation. In *Proceedings of the 5th Intl. Workshop on Worst-Case Execution Time Analysis*, 2005.

[46] Stephan Wilhelm and Björn Wachter. Towards Symbolic State Traversal for Efficient WCET Analysis of Abstract Pipeline and Cache Models. In Christine Rochange, editor, *7th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

Appendix A

Cache Functions

The implementation of the declarations and functions used in the cache analysis can be seen below. The code is modified by the cache-gen tool depending on the cache specification. For instance the function `insert` is modified to call either `cacheMissFIFO` or `cacheMissLRU` and similarly for the `update` function. It should be noted that the function used for cache misses by the FIFO replacement policy is the same as used for LRU.

```
clock x;
address cache[CACHESETS][CACHELINES/CACHESETS];
bool dirty[CACHESETS][CACHELINES/CACHESETS];
int[0,2] write_hit_wait;
```

```
int cache_contents(address adr)
{
    int i;
    address mem_block_offset = adr/BLOCKSIZE*BLOCKSIZE;
    int block_set = mem_block_offset/BLOCKSIZE%CACHESETS;

    if (adr == INVALID_ADDRESS)
        return 1;

    for (i = 0; i < CACHELINES/CACHESETS; i++)
    {
        if (cache[block_set][i] == mem_block_offset)
            return i;
    }

    return 1;
}
```

```
void initialiseDataCache()
{
    int i, j;

    for (j = 0; j < CACHESETS; j++)
    {
        for (i = 0; i < CACHELINES/CACHESETS; i++)
        {
            cache[j][i] =  1;
        }
    }
}
```

```
void cacheHitFIFO(address adr, bool write)
{
    if (write == 1 && write_hit == WRITE_THROUGH)
    {
        write_hit_wait = 1;
    }
}
```

```
void cacheHitLRU(address adr, bool write)
{
    address mem_block_offset = adr/BLOCKSIZE*BLOCKSIZE;
    int block_set = mem_block_offset/BLOCKSIZE%CACHESETS;
    int line = cache_contents(adr);
    int i;
    address tempVal = cache[block_set][line];
    int tempDirty;

    if (write_hit == WRITE_BACK)  //dirty
        tempDirty = dirty[block_set][line];

    for (i = line; i > 0; i  )
    {
        cache[block_set][i] = cache[block_set][i 1];
        if (write_hit == WRITE_BACK)//dirty
            dirty[block_set][i] = dirty[block_set][i 1];
    }

    cache[block_set][0] = tempVal;

    if (write == 1)
    {
        dirty[block_set][0] = 1;

        if (write_hit == WRITE_THROUGH)
            write_hit_wait = 1;
    }
}
```

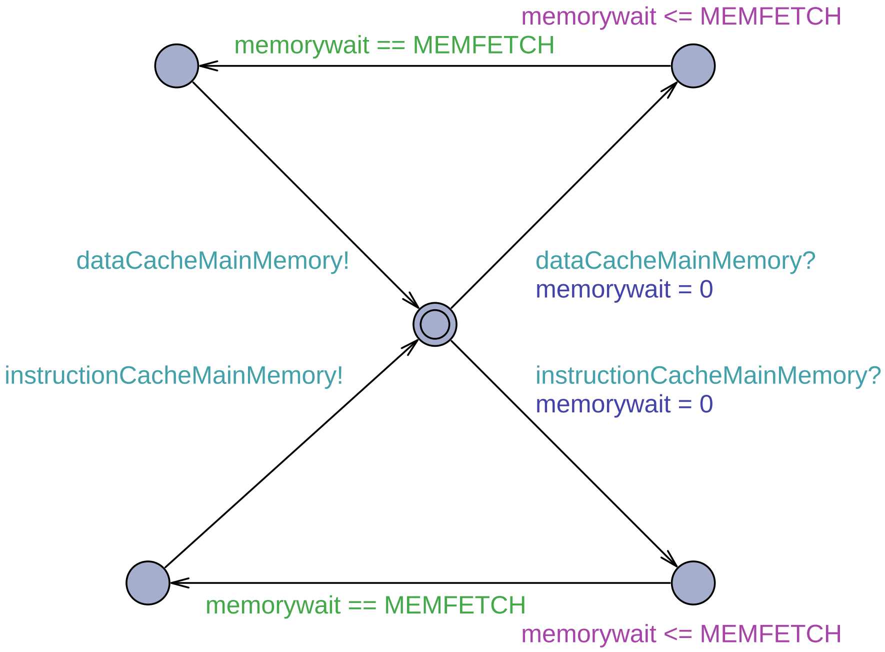
```
void cacheMissLRU(address adr, bool write)
{
    int i, j;
    address mem_block_offset = adr/BLOCKSIZE*BLOCKSIZE;
    int block_set = mem_block_offset/BLOCKSIZE%CACHESETS;

    write_hit_wait = 1;

    if (write_hit == WRITE_BACK and
            dirty[block_set][CACHELINES/CACHESETS 1] == 1)
        write_hit_wait += 1; //dirty

    if (adr != INVALID_ADDRESS)
    {
        for (i = CACHELINES/CACHESETS 1; i > 0; i  )
        {
            cache[block_set][i] = cache[block_set][i 1];
            dirty[block_set][i] = dirty[block_set][i 1];
        }

        cache[block_set][0] = mem_block_offset;
        dirty[block_set][0] = write;
    } else
    {
        for (i = CACHESETS 1; i >= 0; i  )
        {
            for (j = CACHELINES/CACHESETS 1; j > 0; j  )
            {
                cache[i][j] = cache[i][j 1];
                dirty[i][j] = dirty[i][j 1];
            }

            cache[i][0] = INVALID_ADDRESS;
            dirty[i][0] = write;
        }
    }
}
```

```
void update(address adr, bool write)
{
    cacheHitFIFO(adr, write)
}
```

```
void insert(address adr, bool write)
{
    cacheMissFIFO(adr, write);
}
```

Appendix B

Timed Automaton Modelling Main Memory



Thesis Summary

In modern society, use of advanced embedded systems gives rise to the use of real-time systems (RTSs). A RTS typically needs careful scheduling of software processes, which are responsible for providing the intended service of the system. For efficient and careful scheduling of RTSs, safe and sharp worst-case execution times (WCETs) of the involved processes are needed.

To find safe and sharp WCETs, a static WCET analysis of the processes is needed. To be able to find safe and sharp WCETs, analyses need a deep knowledge of the target hardware platform the processes will be executed on. It has, however, proven hard to design modular and efficient WCET analyses which still provides sharpWCETs.

Some of the hardware features that make WCET analysis hard is caching and pipelining. We have studied these features, and ways of incorporating them into a WCET analysis. We have chosen a typical modern RTS processor as the target platform of our prototype, the ARM920T 32 bit processor.

In this thesis the modular WCET analysis method, Modular Execution Time Analysis using Model Checking (METAMOC), is presented based on combining model checking and static analysis. The analysis is divided in four sub-analyses: value analysis, cache analysis, pipeline analysis and path analysis. The value analysis is based on static program analysis, using the formalism of weighted push-down systems (WPDSs). The other analyses all produce models, which separately simulate each part of the hardware platform. These models are then combined, and the combined model is model checked using the UPPAAL model checker.

The METAMOC method takes as input: a binary executable of the program to be analysed (typically from the end-user), and a specification of the cache and a model of the pipeline (from the hardware vendor or researchers). The method produces a model of the process' control flow graph (CFG) from the binary executable, and this CFG is then annotated with the effect of individual instructions, as well as information derived by the value analysis. The model checking of the combined model simulates an abstract execution of the process, given the models of the hardware platform. The model checker finds the longest time-wise path through the process by exploring all paths, and states, as defined by the sub-analyses. This value is an overapproximation of the WCET.

The modularity of the METAMOC method enables us to have different efficiency/sharpness trade-offs, e.g. by having different cache models. The user can experiment with how the process would behave, given a different hardware platform.

Experiments, on WCET benchmark programs from the Mälardalen Real-Time Research Center, have been performed using the prototype implemen-

tation. The experiments show that the METAMOC method gives noticeable improvements compared to traditional, overly pessimistic WCET analyses. Furthermore the benchmarks show that most of the benchmark programs can be verified in less than five minutes. We have found that integrating cache analysis gives up to 97% sharper WCETs, and is thus very worth-while.