UNIVERSIDADE DA BEIRA INTERIOR
Covilhã | Portugal

Timing Analysis - From Predictions to Certificates

Nuno Miguel Pires Gaspar

Submitted to University of Beira Interior in candidature for the degree of
Master of Science in Computer Science

Supervised by Simão Melo de Sousa
Co-Supervised by Rogério Reis

Departamento de Informática
University of Beira Interior
Covilhã, Portugal
http://www.di.ubi.pt

Acknowledgements

I always liked the acknowledgement chapter. In spite that they all follow, more or less, the same structure (thank your advisors, thank your lab colleagues, thank your family), they always tell you something more about the author. I suppose this is natural, in view of the fact that this is most likely the chapter that is less often read carefully, and thus the author feel the freedom to get creative without fear. Also, this is the only place where you can be biased. Very few keep it simple and straight to the point. Inside jokes, clever remarks, Asian sentences, you name it. I have seen it all! Me? Well, I just wrote this paragraph.

First of all, I would like to thank my Advisor, Professor Simão Melo de Sousa, for being the one who first got me into this challenging but fascinating world of Formal Methods. Only time will tell if I will make a career out of it. To my Co-Advisor, Professor Rogério Reis, for always being very friendly, and for all the extensive (and painful. Painful, in a good way!) reviews.

I should also thank all my colleagues from University of Beira Interior. In particular, the ones from the RELEASE - *RELiablE And SEcure Computation Group*, Carlos Carloto, Diogo Fialho, Vasco Nicolau and Joaquim Tojal. Their presence made the *hard* days more tolerable. Oh, scratch that, I meant enjoyable.

To the PhD-wannabes I met in the context of the RESCUE - *REliable and Safe Code execUtion for Embedded systems* project, David Pereira and Vítor Rodrigues. For being very friendly, and always willing to help out.

Last, and definitely not least, to my Mom. For supporting me all these years, providing me the education that I have today. Also, for her love and patience in my, too many often, absent days.

Abstract

In real-time systems timing properties must be satisfied in order to guarantee that deadlines will be met. In this context, the calculation of the worst-case execution time (WCET) is of paramount importance for schedulability analysis. However, this problem can be difficult if the underlying architecture possesses features like caches and pipelines.

This thesis presents all the necessary steps for the safe and precise WCET calculation. We focus ourselves in the use of static analysis-based methods, and in the ARM architecture as target platform. Moreover, in order to ensure the correctness of our calculation to a program consumer, we produce a certificate (or *proof*) whose validity entails compliance with the calculated WCET. This evidence permits to locally validate the calculated WCET, avoiding the need of a blind confidence on the producer.

Keywords

Timing Analysis; Worst-Case Execution Time; Static Analysis; Fixpoint computation; Abstract Interpretation; Abstraction-Carrying Code

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems can be seen as sets of tasks, that are expected to perform some functionality under predefined timing constraints. In general, in order to ensure a correct system behaviour (schedulability analysis), an upper bound estimative for the Worst-Case Execution Time (WCET) of each task is required.

To improve its performance, modern processors include mechanisms like *memory caches*[1] and pipelines which make instruction's execution time context dependent, increasing the difficulty of static timing analysis. To cope with this, one could be tempted to always assume the local worst case scenario (e.g. cache miss) in order to obtain safe predictions, but two problems could arise of such approach. On one hand, it could lead to an excessive over-approximation of the actual WCET, resulting in a waste of hardware resources, and since most modern real-time systems are mass-produced, the determination of tight predictions can lead to production costs cut [1]. By other hand, due to *timing anomalies* [2], the obtained prediction could in fact, be unsafe (an under-approximation).

1.1 Motivation

The determination of safe and tight upper bounds for the WCET has been the object of intensive study in the literature [3], yet, to the best of our knowledge, there is no attempt to provide some warranties of the correctness of the predicted WCET.

Previous works related with the derivation of execution time bounds embrace the use

[1] In the following, only the term cache is used for simplicity.

of formal methods [4, 5], inherently increasing the confidence that one can put on its approach. Furthermore, the use of sound techniques like *Abstract Interpretation* [6] has already made its way into the industry [7], proving to be well suited for the determination of tight, but safe, execution time bounds.

However, in the context of mobile code safety, which has been progressively gaining notoriety in the sphere of real-time systems [8], both at research and industry levels, since they represent an enabling technology to tackle the limitations of standard client-server based approaches, in spite of the fact that previous methodologies are leveraged by the use of a formal approach, one would still have to put his faith on a potential *untrusted* third party, without being able to independently validate the correctness of the predicted WCET.

For instance, embedded systems such as coral sensors or control computers for satellites, which cannot be easily reachable, would highly profit from mobile real-time code [9]. In fact, even a software/system update can be seen as mobile code. Thus, being able to independently validate its timing behaviour would be of major interest for such systems. Moreover, this could also be a valuable asset for original equipment manufacturers and sub-contractors applications.

This lack of an independent validation process is the issue that we address in this work.

1.2 Approach

The general framework of our proposal is illustrated by Figure 1.1. The overall project in which this thesis is included in, has as starting point the extension of the C programming language with annotations. This is accomplished by following a *Design-by-Contract* approach, that define the intended timing properties for each function. The timing specification of the main function is of most importance, however one may also want to define some constraints on the auxiliary functions. Moreover, by placing these annotations directly into the source code, we are also able to express valuable informations for the subsequent timing analysis, such as infeasible paths and loop bounds. This is achieved by the use of a WCET-aware compilation process, targeting the ARM instruction set [10], that preserve the annotations semantics.

Due to the intricacies of modern processors, only at the hardware level one can accu-

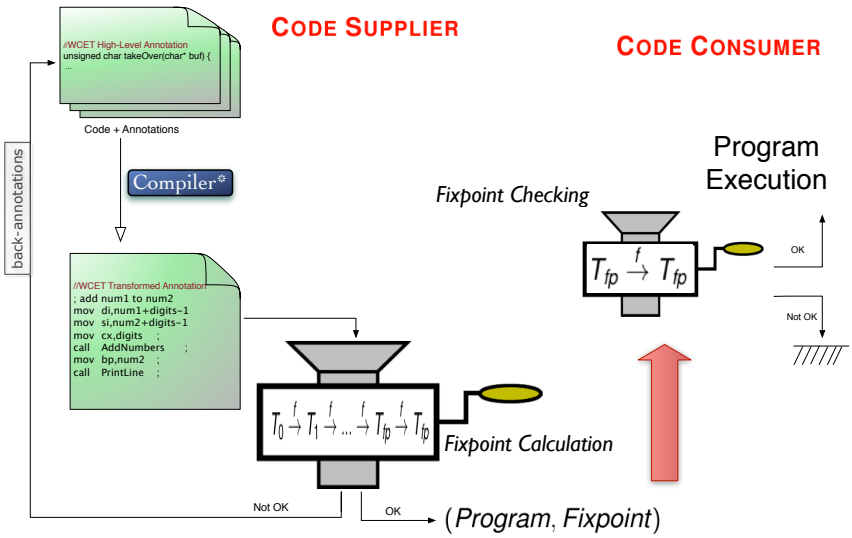


Figure 1.1: ACCEPT: Abstraction-Carrying CodE Platform for Timing validation

rately perform timing analysis [3]. However, feedback about the compliance of the given timing specification should be done at source-code level. Hence, in order to perform timing validation w.r.t. the functions' timing specification, we perform the timing analysis at machine-code level, taking into account the effects of the hardware specific features, and alert of any possible non-compliance through to use of *back annotations* [11]. The idea of this mechanism is to propagate the timing information back to the source-code level, warning the system developers about the violation w.r.t. the timing specification. However, details on this mechanism are out of the scope of this work and will be reported elsewhere.

The concept of Abstraction-Carrying CodE Platform for Timing validation (ACCEPT) is that a potential *untrusted* program producer supplies the code augmented with a certificate. This certificate will allow a program consumer to validate the received code w.r.t. to its timing behaviour.

1.2.1 Abstract Interpretation

Let the set of execution times of a program P be denoted by $[\![P]\!]$. The problem of verifying the compliance of the given timing specification can thus be formulated as follows:

$$P \text{ respects the timing specification } \mathscr{I} \text{ if } [\![P]\!] \subseteq \mathscr{I},$$

where $\mathscr{I}$ stands for the intended timing behaviour, i.e., the set of accepted execution times. The idea is to express the set of concrete states occurring at each program point, i.e. the *collecting semantics*[2] [12], of P as the fixpoint of a set of recursive equations.

In general, however, the state space to be considered is too large to exhaustively explore all possible executions and some abstraction of the application domain is required in order to make the timing analysis feasible. With this in mind, our approach relies on abstract interpretation [6] as the underlying technique. With its use, not only it provides us an adequate framework to reason about, but also with an elegant way to infer an abstract model of the program that can play the role of a certificate.

In the abstract interpretation framework, a program P is interpreted over a simpler *abstract domain* $\mathscr{D}_\alpha$. This abstract domain permits to trade efficiency over precision, i.e., although it is an approximation, by computing the fixpoint over this abstract domain, we will be able to produce precise, yet safe, (over-)approximations of the collecting semantics.

The fixpoint calculation over this abstract domain will allow us to safely predict the processor behaviour for the program's execution (e.g. pipeline stall). With that information, and following the approach from the standard WCET architecture [13], one can compute the execution times of the program's *basic blocks*[3], and then obtain the WCET and Best-Case Execution Time (BCET), by determining their paths through the control-flow graph. This is achieved by solving the corresponding *integer linear program*, maximizing it for execution time in case of WCET computation, and analogously for BCET computation.

Let $[\![P]\!]_\alpha$ be the set of execution times calculated over the abstract domain $\mathscr{D}_\alpha$. It is clear that $[\![P]\!]_\alpha$ is lower- and upper-bounded by BCET and WCET, respectively. Moreover, since the comparison between actual and intended semantics is easier if done in the same domain, we assume that the intended timing specification is also given in the abstract domain, i.e., $\mathscr{I}_\alpha \in \mathscr{D}_\alpha$.

The problem of verifying the compliance with the given timing specification can now be reformulated as:

$$P \textit{ respects the timing specification } \mathscr{I}_\alpha \textit{ if } [\![P]\!]_\alpha \subseteq \mathscr{I}_\alpha.$$

[2] Also called *static semantics* in the literature [6].

[3] A basic block is a maximal sequence of straight-line code in the program, i.e., it has one entry point and one exit point, without any branches contained within it.

At this stage, feedback regarding any possible non-compliance of P w.r.t. the timing specification can be reported through the use of *back-annotations* [11], permitting the system developers to proceed accordingly.

1.2.2 Abstraction-Carrying Code

Proof-Carrying Code (PCC) [14] is a general mechanism enabling a program consumer to locally check the validity of the code w.r.t. some safety policy. The inherent key benefit is that there is no need to trust any third party. However, there are three essential challenges for PCC to be used in practice:

(i) definition of *expressive safety policies*,

(ii) automatic generation of the certificate, i.e., proving the program correct, and

(iii) efficient certificate checking in the consumer side.

In the context of mobile code safety, most approaches rely on theorem proving, whereas Abstraction-Carrying Code (ACC) [15] relies on abstract interpretation.

In ACC, and in particular for the purposes of our platform, the above challenges are addressed by (i) getting hold of the effects that the processor specific features (e.g. caches, pipeline) have on the execution time, which has already been addressed in the literature [16, 17]; (ii) using a fixpoint static analyser to automatically infer an abstract model of the program, which can be then used as a certificate; and (iii) by a simple, easy-to-trust fixpoint checker.

As mentioned earlier (see Subsection 1.2.1), this fixpoint computation will yield the execution times of the program's basic blocks, and the remaining task of calculating the BCET and WCET being achieved by means of integer linear programming techniques. However, we can also let this fixpoint play the role of a certificate. The idea of ACC, is that when a consumer receives a program, it is augmented with a certificate. Since the certificate is supposed to be a fixpoint, another iteration over it cannot produce any changes. Thus, if it is a fixpoint, then the consumer can independent and locally compute the BCET and WCET, and check if the program complies with the intended timing behaviour.

1.3 Contributions

The work reported here is included in a broader effort to provide a source level feedback on a BCET and WCET computation platform with certificate generation in the context of mobile code. This platform considers an high-level annotation language, preserving its semantics through a *WCET-aware* compilation process, and a *back-annotation* mechanism [11]. However, these features are not a subject of this work, and will be discussed elsewhere. Here, we focus on the low level interface. In this sense, this work reports on the certificate generation and validation, and intends to introduce and justify the underlying architecture.

The main contributions of this work are the following:

- An open-source architecture for BCET and WCET computation that follows state of the art algorithms for its implementation.
- The first application of the *Abstraction-Carrying Code* concepts to the static timing analysis field.

The following publications resulted in the context of the work developed in this thesis:

- *Diogo Fialho, Nuno Gaspar, Jorge Sousa Pinto, Rogério Reis and Simão Melo de Sousa.* **Worst-Case Execution Time: From Predictions to Certificates**. *Proceedings of the Days In Logic, DiL'2010*, Porto, Portugal, 2010
- *Diogo Fialho, Nuno Gaspar, Jorge Sousa Pinto, Rogério Reis and Simão Melo de Sousa.* **Towards a Worst-Case Execution Time Platform with Certificate Production**. *Supplement to the Proceedings of the 8th European Dependable Computing Conference, EDCC'2010*, Valencia, Spain, 2010
- *Nuno Gaspar, Rogério Reis and Simão Melo de Sousa.* **Timing Analysis - From Predictions to Certificates**, *Proceedings of INForum'10 – Specification, Verification, and Testing of Critical Systems, EVTSIC'10*, Braga, Portugal, 2010, (*to appear*)

1.4 Organization of this thesis

The remainder of this thesis is organised as follows. Chapter 2 presents an overview of the ARM 7 architecture. The main ingredients of real-time systems and timing analysis

are discussed in Chapter 3. The theory underlying the tool developed in the context of this thesis is discussed in Chapter 4. Chapter 5 presents this tool. Finally, conclusions and directions for future work are discussed in Chapter 6.

Chapter 2

The ARM Architecture

In this Chapter we present an overview of the ARM 7 architecture that was the target for our timing analysis platform. The instruction set and the major features of the considered processor, ARM7TDMI-S, are shown and discussed.

2.1 Main Characteristics

The ARM7TDMI-S processor is a member of the ARM family of general-purpose 32-bit microprocessors, known for their high performance while keeping a very low-power consumption and gate count. It is characterised by being an highly popular Reduced Instruction Set Computer (RISC) architecture. It is very common in the market, being broadly incorporated in widespread embedded systems such as the Game Boy Advance, Nintendo DS or the Apple iPod.

Although being a 32-bit microprocessor, it is also capable of running a 16-bit instruction set, known as *Thumb* [10], and is actually, the reason why the ARM7TDMI-S processor has a "T" in its name. The Thumb instruction set helps the achievement of a greater code density, and in fact, can be seen as a compressed form of a subset of the ARM instruction set. It is usual to mix the ARM and Thumb instruction sets in a single application, this flexibility allows to use one instruction set or another on a routine-by-routine basis. In order to determine how the instructions stream is to be interpreted, whether in ARM or Thumb state, the processor examines the 5^{th} bit, known as the T bit, of the CPSR register. If the T bit is set, the instruction stream is interpreted as 16-bit Thumb instructions, otherwise is interpreted as 32-bit ARM instructions.

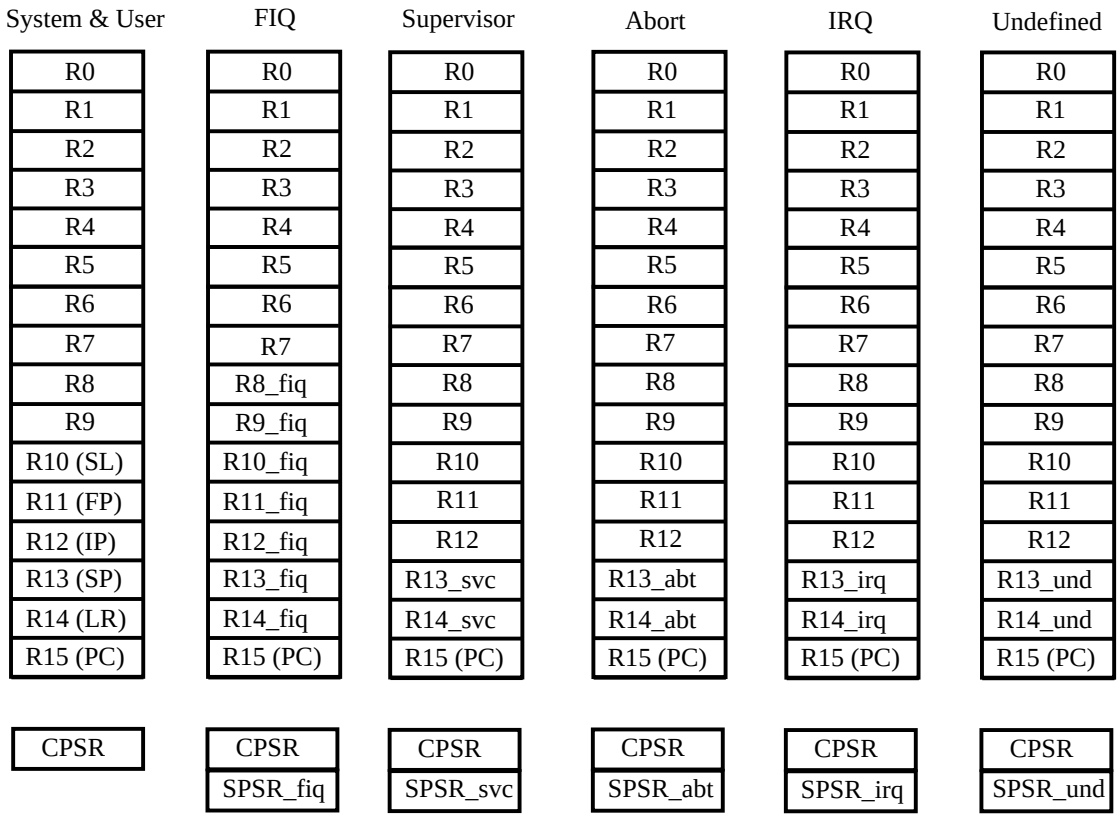


Figure 2.1: Acessible Registers in the different operating modes

In ARM state, whenever operating in System or User mode, as illustrated in Figure 2.1, there are 16 general registers and a status register accessible at any time. The general registers range from R0 to R15, with the last six having alias names. Three of this last six having special roles[1]. For instance, R13 is also known as the Stack Pointer (SP), R14 as the Link Register (LR), and R15 as the Program Counter (PC). The remaining 13 registers have no special hardware purpose. Their uses are defined solely by software.

Similarly to the state, either ARM or Thumb, the contents of the CPSR register also determines the operating mode, and thus the accessible registers. Moreover, besides holding the bit that determines how the instruction stream is to be interpreted (i.e. either in ARM or Thumb state), and the operating mode, it also contains other flags, that, in fact, are particularly relevant for the purposes for this work. The user-configurable bits of the CPSR register are depicted in Figure 2.2. In order to serve our interests, we will be

[1]The remaining three registers, in descending order, are aliased: Intra-Procedure call scratch register, Frame Pointer and Stack Limit, respectively.

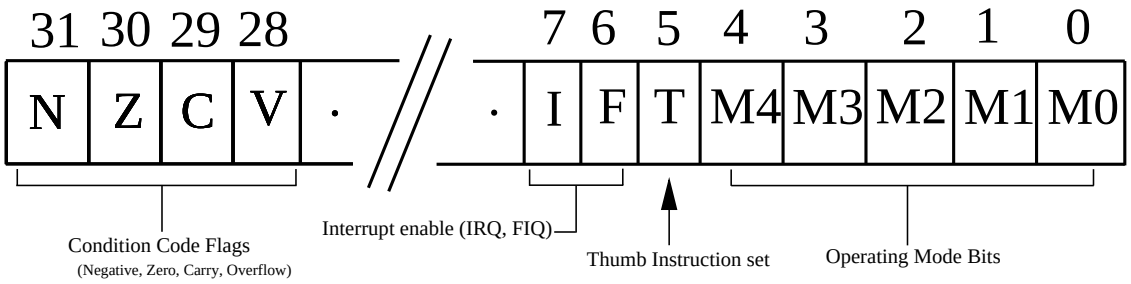


Figure 2.2: CPSR register user-configurable bits

devoting more attention to the *Condition Code Flags*.

The N, Z, V and V (meaning Negative, Zero, Carry and oVerflow, respectively) bits are collectively known as the condition code flags, often referred simply as flags. As we shall see in section 2.2, in the ARM Assembly language all instructions can be conditionally executed. Hence, the purpose of these flags is to determine whether an instruction is to be executed or ignored.

2.2 ARM Instruction set

The ARM instruction set summary is given in table 2.1.

As mentioned earlier, in the ARM Assembly language the instructions can be conditionally executed. This is achieved by appending a suffix to the instruction's mnemonic. The possible suffixes are known as the *condition codes*, and are listed in table 2.2.

As shown in table 2.2, there are fifteen condition codes. Indeed, the last one always evaluates to *true*, meaning that one instruction without a condition code, and the same suffixed with the condition code AL, are semantically equivalent. For the sake of clarity, let us consider the following two examples.

(1) `MOVAL r1 #5`
(2) `BEQ LABEL`

In example (1), the condition code AL always evaluates to *true*, thus, this instruction is semantically equivalent to `MOV R1 #5`. However, in (2) the instruction would only be executed if the CPSR's Z bit would be set.

Furthermore, the *Data Processing* and *Multiply and Multiply-Accumulate* instructions can also be suffixed by S (for instance, `ADDS R1 R3`). If this is the case, it means that

Mnemonic	Instruction	Action
	Branch and Exchange	
BX	Branch and Exchange	$R15 := Rn, Tbit = Rn[0]$
	Branch and Branch with Link	
B	Branch	$R15 := address$
BL	Branch with Link	$R14 := R15, R15 := address$
	Data Processing	
MOV	Move register or constant	$Rn := Op2$
MVN	Move negative register	$Rn := OxFFFFFFFF$ xor $Op2$
CMP	Compare	$CPSRflags := Rn - Op2$
CMN	Compare Negative	$CPSRflags := Rn + Op2$
TEQ	Test bitwise equality	$CPSRflags := Rn$ xor $Op2$
TST	Test bits	$CPSRflags := Rn$ and $Op2$
AND	AND	$Rd := Rn$ and $Op2$
EOR	Exclusive or	$Rd := Rn$ xor $Op2$
SUB	Subtract	$Rd := Rn - Op2$
RSB	Reverse subtract	$Rd := Op2 - Rn$
ADD	Add	$Rd := Rn + Op2$
ADC	Add with carry	$Rd := Rn + Op2 + Carry$
SBC	Subtract with carry	$Rd := Rn - Op2 - 1 + Carry$
RSC	Reverse Subtract with carry	$Rd := Op2 - Rn - 1 + Carry$
ORR	OR	$Rd := Rn$ or $Op2$
BIC	Bit clear	$Rd := Rn$ and not $Op2$
	PSR Transfer	
MRS	Move PSR status/flags to register	$Rn := PSR$
MSR	Move register to PSR status/flags	$PSR := Rm$
	Multiply and Multiply-Accumulate	
MUL	Multiply	$Rd := Rm * Rs$
MLA	Multiply and Accumulate	$Rd := Rm * Rs + Rn$
	Single Data Transfer	
LDR	Load register from memory	$Rd := address$
STR	Store register to memory	$address := Rd$
	Block Data Transfer	
LDM	Load multiple registers	Stack Manipulation (Pop)
STM	Store Multiple	Stack Manipulation (Push)
	Single Data Swap	
SWP	Swap register with memory	$Rd := [Rn], [Rn] := Rm$
	Software Interrupt	
SWI	Software Interrupt	OS Call

Table 2.1: ARM Instruction Set relevant for the purposes of our work

Suffix	Flags	Meaning
EQ	Z set	equal
NE	Z clear	not equal
CS	C set	unsigned higher or same
CC	C clear	unsigned lower
MI	N set	negative
PL	N clear	positive or zero
VS	V set	overflow
VC	V clear	no overflow
HI	C set and Z clear	unsigned higher
LS	C clear or Z set	unsigned lower or same
GE	N equals V	greater or equal
LT	N not equal to V	less than
GT	Z clear AND (N equals V)	greater than
LE	Z set OR (N not equal to V)	less than or equal
AL	(ignored)	always

Table 2.2: Condition Codes

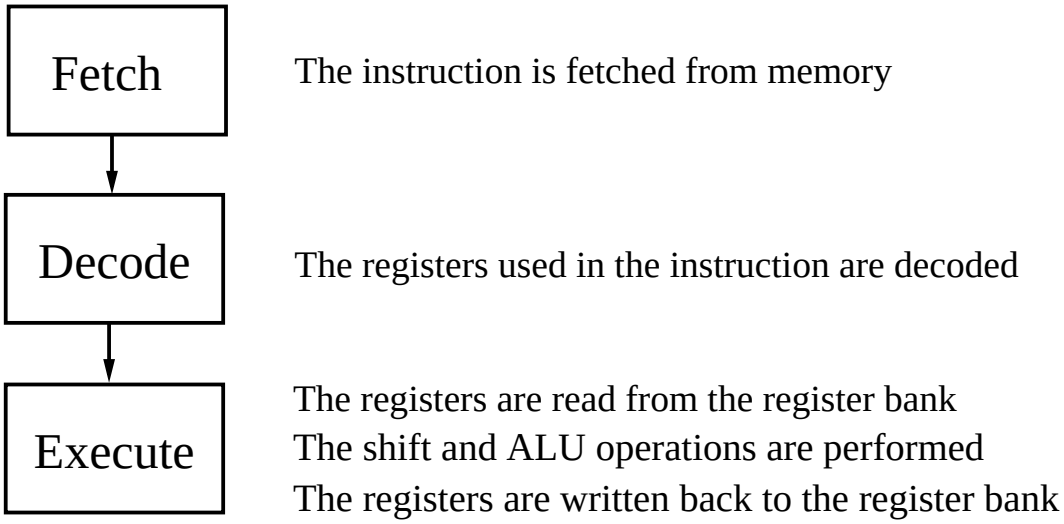


Figure 2.3: The instruction pipeline

the CPSR register is to be affected accordingly. For the sake of clarity, let us consider the following example:

$$2^{31} - 1 + 1 = -2^{32}$$

This operation provokes an overflow and produces in a negative result, thus the CPSR register would be affected as follows: NZCV = 1001.

The reader may wonder the significance of the presence of such suffix, S, in the instructions CMP, CMN, TEQ and TST. These instructions already affect the CPSR register, thus the presence or absence of the S suffix is irrelevant, since it is implied anyway.

2.3 Pipeline

The ARM7TDMI-S processor features a 3-stage pipeline, allowing the overlapping of the instruction's execution, and thus better performance. As shown in Figure 2.3, the three stages of the pipeline are *Fetch*, *Decode* and *Execute*. The hardware of each stage is designed so that each stage is independent, and thus, up to three instructions can be processed simultaneously.

The basic idea of instruction pipelining is to split the processing of an instruction into a series of independent steps, and thus, different steps from different instructions can be

overlapped in the pipeline. For the sake of clarity, let us consider the following example, where time runs horizontally, with each tick of time corresponding to a processor cycle, and the pipeline stages are represented on the vertical axis. An instruction **A** takes 3 cycles to execute, and an instruction **B** takes 4 cycles:

Fetch	A		
Decode		A	
Execute			A

Fetch	B			
Decode		B	B	
Execute				B

Without instruction pipelining, the execution of **AB** would take 7 cycles, however, by overlapping **A** and **B** we obtain a 5 cycles execution:

Fetch	A	B			
Decode		A	B	B	
Execute			A		B

Early approaches to timing analysis assumed context independence, ignoring the execution history [3]. However, as seen in this trivial example, this assumption is no longer true for modern processors featuring pipelines.

Further, the performance of the pipeline can be measured by the Clock Cycles per Instruction (CPI). In the optimal case, as soon as the pipeline has been completely filled, the execution of an instruction takes one cycle. In such cases, we say that the CPI is equal to 1.

2.3.1 Instruction Cycle Count Summary

In the ARM7TDMI-S pipeline while one instruction is being fetched, the previous is being decoded, and the one prior to that is being executed. Table 2.3 presents the number of cycles required by an instruction, when the *Execute* stage is reached.

In table 2.3,

- **n** is the number of words transferred
- **m** is:
  - 1 if bits [32:8] of the multiplier operand are all zero or one.
  - 2 if bits [32:16] of the multiplier operand are all zero or one.

Instruction	Qualifier	Cycle count
Any unexecuted	Condition codes fail	1
Data processing	Single-cycle	1
Data processing	Register-specified shift	2
Data processing	R15 destination	3
Data processing	R15, register-specified shift	4
MUL	-	m + 1
MLA	-	m + 2
MULL	-	m + 2
MLAL	-	m + 3
Branches	-	3
LDR	Non-R15 destination	3
LDR	R15 destination	5
STR	-	2
SWP	-	4
LDM	Non-R15 destination	(n-1) + 3
LDM	R15 destination	(n-1) + 5
STM	-	(n-1) + 3
MSR, MRS	-	1
SWI	-	3

Table 2.3: Instruction cycle counts

- 3 if bits [31:24] of the multiplier operand are all zero or one.
- 4 otherwise.

2.3.2 Pipeline Stalls

Instruction pipelining is a powerful method to enhance performance. However, some situations give rise to *pipeline stalls* which provokes the execution to take extra cycles to complete. Consider the two following instructions **A** and **B**:

Fetch	A							
Decode		A						
Execute			A	A	A	A	A	A

Fetch	B		
Decode		B	
Execute			B

The execution of **A** takes 6 cycles, which will make the execution of **B** in the pipeline stall:

Fetch	A	B							
Decode		A	B						
Execute			A	A	A	A	A	A	B

Circumstances leading to pipeline stalls are denominated *pipeline hazards*. There are thee types of such hazards. (i) *Structural hazards* arise from resource conflicts, (ii) *data hazards* are caused by data dependences between overlapping instructions, and (iii) *control hazards* are due to control flow changes, i.e., branches.

✧ ✧ ✧

In this chapter we presented the essential features of the ARM7TDMI-S processor. The ARM instruction set, the accessible registers in System/User mode and the instruction pipeline were the topics of main focus. For further details regarding the ARM architecture we refer the reader to the information discussed in [10].

In the next chapter we present the basic notions concerning the timing analysis of Real-Time Systems (RTS).

Chapter 3

Real-Time Systems and Timing Analysis

In this chapter we present the main ingredients of Real-Time Systems with emphasis on their timing analysis. The main challenges, methods and tools for timing analysis are presented and discussed.

3.1 Real-Time Systems

RTS are required to satisfy stringent timing properties. These, are often defined as output computations responses to some input stimuli, that need to be completed under some timing interval. In general, in order to guarantee that the imposed deadlines will be met, an upper bound on each task's execution time is necessary. However, determining the execution time on any arbitrary program is *undecidable*, otherwise one could solve the *halting problem* [18]. Hence, a restricted form of programming is typically employed, recursion and loops bounds are explicitly bounded, and thus ensuring termination.

The tasks that compose RTS characteristically present variations on their execution times depending on many aspects. Hardware features, like caches and pipelines, and input data are some of the elements that make the execution time context dependent. To the longest time of all possible executions we call it WCET, and analogously BCET to the shortest.

Timing Analysis is the process of deriving execution time bounds or estimates. In Figure 3.1 its basic concepts are presented. The set of execution times is upper- and lower-bounded by WCET and BCET, respectively. In general, due to the complexity of modern processors, and the large program's state space, the actuals WCET and BCET

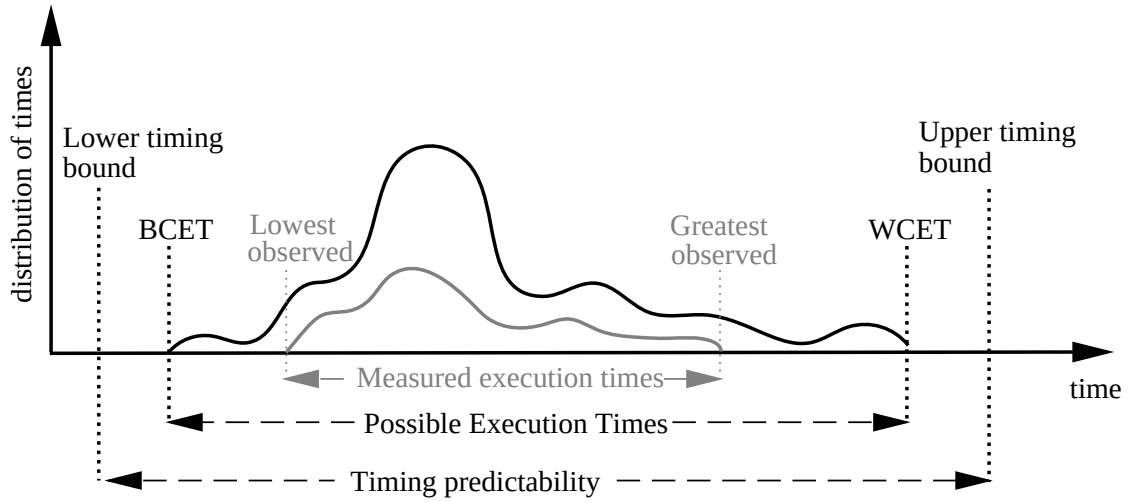


Figure 3.1: Basic concepts of Timing Analysis

cannot be determined. These characteristics of the overall system, affecting the timing analysis, form the notion of *timing predictability*.

The two main criteria to assess a particular method for timing analysis are *safety* and *precision*. The former is directly related with whether the method produces bounds or estimates, i.e., if it produces an upper-bound for the WCET, and thus is safe, or if the predicted value can in fact be lower than the actual WCET (and vice-versa for BCET), and thus unsafe. The latter, is concerned about how far is the prediction of the actual WCET/BCET. In general, only static analysis-based methods provide guarantees w.r.t. the safety criteria. Measurement-based methods cannot ensure that the measured executions capture the WCET and BCET. These usually underestimate the WCET and overestimate the BCET, and thus are unsafe.

3.1.1 Timing Anomalies

Most powerful microprocessors suffer from *timing anomalies* [19]. These are contra-intuitive influences of the (local) execution time of one instruction on the (global) execution of the whole task [3]. For instance, tacitly assuming that a cache-miss penalty yields a longer execution time than a cache hit is wrong for processor with timing anomalies. In Figure 3.2[1] a *speculation-caused anomaly* is depicted.

While intuition would suggest that a cache hit would lead to a shorter execution time,

[1] Figure 3.2 was taken from [2]

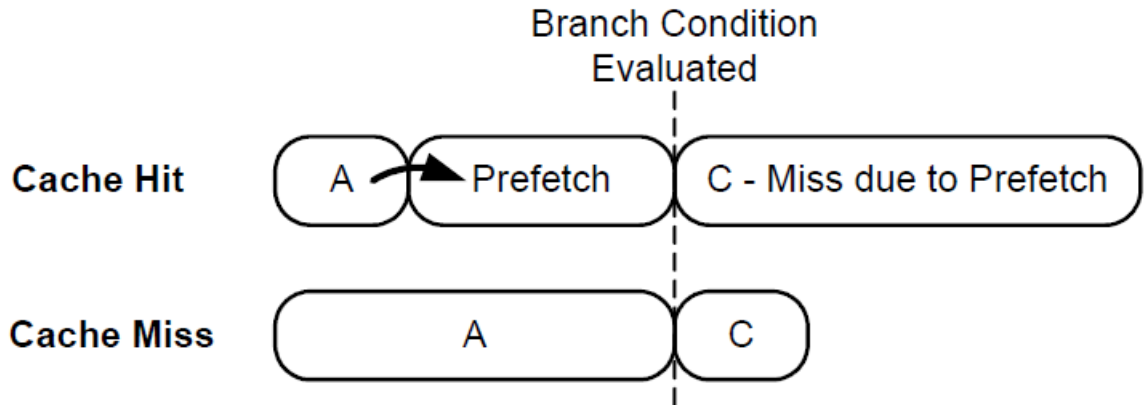


Figure 3.2: Timing Anomaly caused by speculation

in [20] it has been shown that this need not be true. This was observed on the Motorola ColdFire 5307 processor. The cause of this anomaly arises from the speculation on the outcome of conditional branches, i.e., it *prefetches*[2] instructions in one of the directions of a conditional branch. However, when the condition is evaluated, it might be the case that the processor speculated in the wrong direction. All the effects that the speculation gave rise, have to be undone. Further, the contents of the cache have been damaged, since the wrong instructions have been fetched. Hence, the burden of a branch misprediction surpass the costs of a cache miss.

3.2 Methods and Tools

A lot of research has been dedicated to the safe and precise derivation of the WCET [3]. Indeed, the increasing amount of RTS being used, and their inherent challenges make it a very interesting research topic. There are essentially two classes of methods employed in the timing analysis field: *static analysis-based* methods and *measurement-based* methods. The former class of methods, does not rely on executing code, but rather receive the code itself as input, with or without some extra informations (annotations, user domain-specific knowledge, etc), perform a control-flow analysis, and derive bounds by combining the control flow with some abstract model of the considered hardware. While the latter executes the code, or parts of it, in a given hardware or simulator for some set of

[2]Also denominated *preemptive execution* by Intel, instruction prefetch is a technique to speed up the execution of programs. It occurs when a processor requests an instruction from the main memory before it is actually needed.

input values.

3.2.1 Measurement-based Methods

The current trend in this field is focused on static analysis-based methods. However, there are several tools, commercial and research prototypes, based on measurements. *RapiTime* [21] tries to derive the WCET by measuring how long sections of code take from predefined inputs. It is also known as the commercial quality version of *pWCET* [22]. *SymTA/P* [23] targets C programs running on microcontrollers. It uses an *hybrid approach*, doing path analysis on source code level combined with measurements on object code level. From Sweden, another two research prototypes using an *hybrid approach* are known: the Chalmers prototype [24] and *SWEET* [25]. Both use simulation as the underlying methodology. The former works directly on Power-PC binaries, while the latter integrates a research C compiler in order to perform flow analysis in the intermediate code.

The benefit of measurement-based methods is that they are easier to apply to new target processors, since modelling the processor's behaviour is not necessary. It is also claimed that the produced estimates for the WCET/BCET are more precise then the bounds produced by static analysis-based methods. However, the exacts BCET and WCET cannot be determined in general. Studies regarding precision of estimates or bounds are habitually comparisons with the lowest/greatest observed time from a large, but not exhaustive[3], set of tests [3]. On the other hand, the main difficulty of measurement-based methods is to measure execution times accurately. Obtaining a fine granularity, without affecting the program's behaviour may require a processor-specific solution. A general approach, is to place the code fragment to be measured in a loop with a predetermined bound, measure its execution, and then divide by the number of iterations.

3.2.2 Static Analysis-based Methods

Static analysis-based methods are essentially constituted by the components depicted in Figure 3.3[4].

Further, they can be classified according to the level they operate. All require the pres-

[3] Indeed, this can be seen as the classic Testing Vs Verification debate.

[4] This Figure is taken from [3].

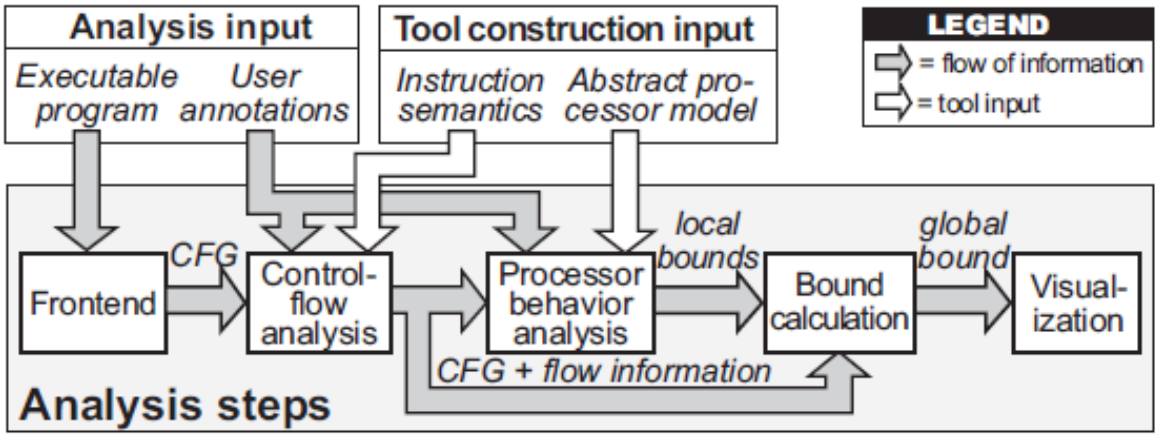


Figure 3.3: Core components of a static timing-analysis tool

ence of a binary file, since only at that level one can obtain all the necessary information for timing analysis, however some also require the source code. *Heptane* [26], *Chronos* [27] and *TuBound* [28] require both the source and object codes. From Vienna University of Technology, there are three research prototypes [29]: *Vienna S*, *Vienna M* and *Vienna H*. While the first one relies mostly on static program analysis, the remaining employ genetic algorithms and model checking, respectively. Only *Vienna M* work solely at the object level, the remaining two also require the presence of the source code. Finally, *aiT* [7] is the state-of-the-art static timing analyser, designed

"*according to the requirements of Airbus France for validating the timing behavior of critical avionics software, including the flight control software of the A380, the world's largest passenger aircraft*".

It operates at the object level, the presence of the source code is not necessary. However, it may also need some user input in order to obtain a result, or to improve the precision of the predicted bound.

The inherent advantage of static analysis-based methods is that running the program is not necessary, which may be particularly relevant for programs that require complex and expensive equipments. Moreover, rather than estimates, these methods produce bounds and thus of crucial importance for hard[5] RTS. On the other hand, its main challenge lies in the modelling of the processor behaviour.

[5]While the strict definition of an hard RTS is that missing the deadline would constitute failure of the system, it is referred in the literature as RTS whose lack of success in meeting a deadline could endanger human lives.

3.3 Related Work

There are numerous approaches, found in the literature, focused in algorithms and tools for the derivation of the WCET [4, 30, 7, 31]. The goal of this work is not to present yet another approach of that type, but rather to emphasize the production of a certificate that can be then used to validate the predicted WCET and BCET.

Nevertheless, we should refer that the problem of certifying resource consumption, namely execution time, as already been addressed before, like in the work of Crary & Weirich [32], that use an extended type system capable of specifying and certifying bounds on resource consumption. However, this work makes no effort to determine bounds on execution times, but rather provides a mechanism to certify those bounds (for instance, obtained via a previous program analysis). The result of their approach is an executable object that is certified w.r.t. resource consumption. Furthermore, Bonenfant et al and Jost et al [33, 34] present an interesting combination of information retrieved at source code level, with low-level timing information gathered with AbsInt's aiT tool [7]. Both works provide guaranteed bounds on worst-case execution times for a strict, higher-order programming language.

The *Mobility, Ubiquity and Security* (MOBIUS) [35], and the *Mobile Resource Guarantees* (MRG) [36] research projects also aim at the certification of resource consumption, their approaches mostly rely on theorem proving, whereas ours relies on abstract interpretation.

Moreover, in [37], Kwon et al, present a safe mobile code representation for Real-Time Java programs. Based on a *Static Single Assignment* (SSA) representation, they argue that their approach is able to cope with the intrinsic challenges of high-integrity real-time software (such as subsystems upgrades and software portability). Unlike prior approaches to theirs [38], they focus more on temporal predictability and safety of Java programs.

In this chapter we presented the basic notions concerning the timing analysis of RTS. Its challenging aspects, methods and tools were discussed.

In the next chapter we present the core of the work developed in the context of this thesis.

Chapter 4

From Predictions to Certificates

In this Chapter we present the work developed in the context of this thesis, where the mathematical formalisms will be introduced as needed.

In Section 4.1 the process of reconstructing the control-flow graph from an ARM assembly file is shown. From the resulting control-flow graph, in Section 4.2 we present our efforts towards a pipeline analysis, that is supported by the sound theory of Abstract Interpretation. Section 4.3 demonstrates how to compute bounds on execution times. Sections 4.4 and 4.5 discuss the production of the certificate, and how it can be used for validation purposes, respectively.

4.1 Control-Flow Graph Analysis

A Control-Flow Graph (CFG) is standard representation in program analysis, using a graph notation, it expresses all the paths that a program can have during its execution. Usually, the actual control flow of a program is not fully known due to complex control transfers such as computed branches. Hence, when reconstructing a CFG, it is common to start by a safe (over-)approximation, and then try to refine it by means of static analysis.

The control flow of a program can be described completely by its Interprocedural Control-Flow Graph (ICFG) [39]. It is composed by two elements:

- A Call Graph (CG) describing the relationships between the program's procedures[1].

[1] Also addressed as *procedures* or *functions* in the literature.

The nodes stand for procedures, while the edges entail procedure calls.

- A Basic Block Graph (BBG) describing the *intraprocedural* control flow for each of the program's procedures.

The more precise the ICFG is, the better the timing analysis can perform. However, more importantly, in the context of real-time systems it is imperative that the ICFG is *safe* under all circumstances. Removing an edge without certainty that it represents a path that will never be taken by the program, could seriously compromise the timing analysis. For instance, if the removed edge was in fact, part of the path that led to longest time of execution, then our WCET prediction would be an under-approximation, and thus unsafe.

4.1.1 Interprocedural Control-Flow Graph Reconstruction

The process of reconstructing the ICFG is separated in two phases. First the CG is built by gathering the called procedures, and then we proceed by constructing the BBG of each of them. In ARM, procedure calls are made by means of the *branch with link* instruction, BL (see Table 2.1). Hence, gathering the called procedures is a matter of collecting the targets of these branches, and perform analogously in the targeted procedures.

Having the CG, it is time to build the BBG of each of the called procedures. There are two possible approaches for reconstructing the control flow from a sequential list of assembly instructions: a *top-down* approach, or a *bottom-up* approach. Our methodology follows a *top-down* approach, since it the most natural, and allows to cope more efficiently with control flow branches. For instance, consider the following example:

```
B label+7
```

A branch target can be a static label or it can also be an arithmetic operation, and thus coping with these would only require to evaluate the target expression in order to determine the node's successor. On the other hand, in a *bottom-up* approach, one wants to determine the node's predecessors, and thus a prior analysis would be required in order to be able to know which are the nodes predecessors (or add an edge to the successor whenever encountering a branch).

At this stage we are only concerned about obtaining a safe (over-)approximation of the CFG. Hence, whenever in doubt regarding a node's successors, a conservative approach will be taken. For instance, let us consider the C program represented in listing 4.1.

Listing 4.1: Toy Example

```
1  int main(){
2
3  int x;
4
5  x = 5;
6
7  if(x > 6)
8            return 1;
9
10 return 0;
11 }
```

After compiled with GNU compiler for ARM [40], the assembly code presented in listing 4.2 is generated.

Listing 4.2: Compiled ARM Assembly

```
1           mov     ip, sp
2           stmfd   sp!, {fp, ip, lr, pc}
3           sub     fp, ip, #4
4           sub     sp, sp, #8
5           mov     r3, #5
6           str     r3, [fp, #-16]
7           ldr     r3, [fp, #-16]
8           cmp     r3, #6
9           ble     .L2
10          mov     r3, #1
11          str     r3, [fp, #-20]
12          b       .L1
13 .L2:     mov     r3, #0
14          str     r3, [fp, #-20]
```

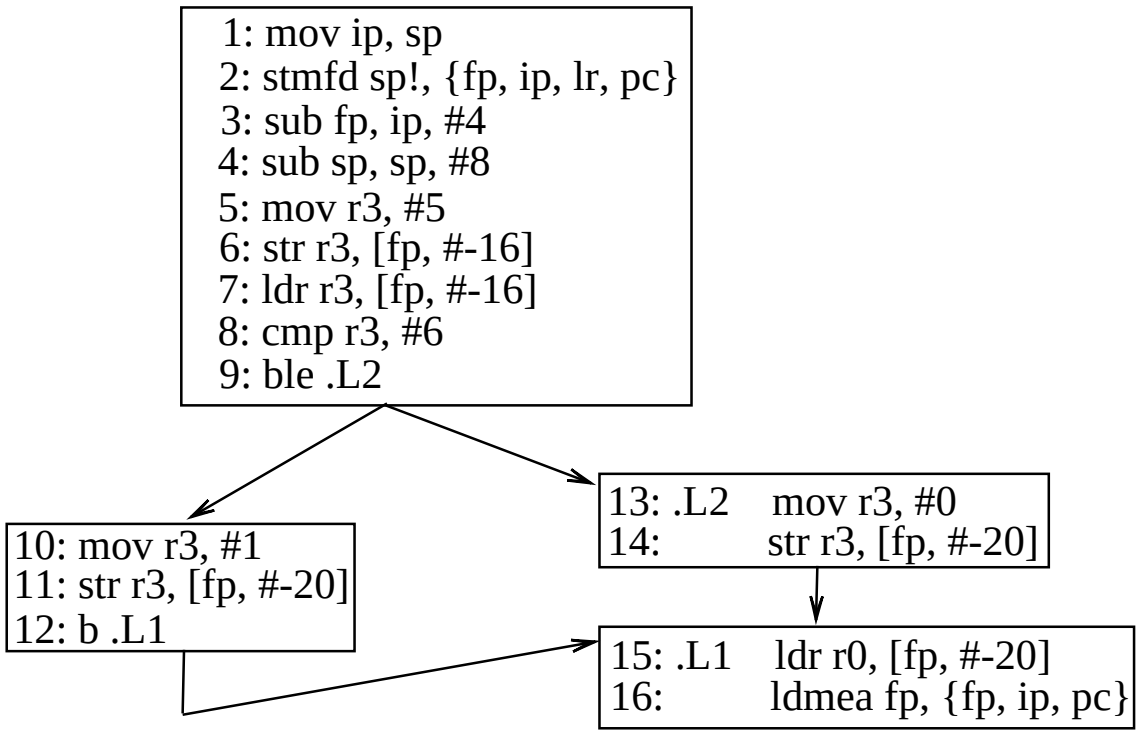


Figure 4.1: Basic Bock Graph

```
15 .L1:     ldr     r0, [fp, #-20]
16          ldmea   fp, {fp, sp, pc}
```

Proceeding to the BBG reconstruction, the obtained control flow is as illustrated by Figure 4.1.

By inspecting the source code of our toy example (listing 4.1), it is clear that the conditional will always evaluate to *false*, and thus the instruction on the 8^{th} line (**return** 1) will never be executed. Inspecting the assembly (listing 4.2) however, is not as straightforward. In the 5^{th} line the value 5 is stored in register *r3* and later compares it with the value 6 ($8^{th} line$). This comparison instruction performs a subtraction as indicated in Table 2.1.

$$R3 - 6 = 5 - 6 = -1,$$

and affects the CPSR flags accordingly:

$$NZCV = 1010.$$

The careful reader may wonder why the C flag (Carry) is set in this operation. This is due to the fact that in subtraction operations, clearing the Carry flag (C=0) means the presence of a *borrow*[2], thus setting the Carry flag means its absence [41], which is the case in this operation.

In the 9^{th} line a conditional branch takes place. The condition to be evaluated, *Less or Equal* (*LE*), will always evaluate to *true*, since, according to Table 2.2, the *LE* condition code is evaluated as follows:

***Z** or (**N** not equal to **V**)* =
false or (**true** not equal to **false**) =
false or **true** =
true,

thus, the program will always branch to the address evaluated by the *.L2* label, where the value 0 is stored in register *R3*, and later stored in register *R0* (15^{th} line).

As mentioned earlier, at this stage the main concern is to produce a safe approximation of the program's control-flow. Having this, an analysis can be performed in order to try to determine the outcome of such conditional branches, and, if possible, safely remove the edges that are never taken during program's execution.

4.1.2 Data-Flow Analysis for Conditional Branches

Data-Flow Analysis (DFA) is a well established technique in the area of compiler construction and used to obtain information about a program that holds for all executions of the program. Typically, it is achieved by setting up *data-flow equations* and computing a fixpoint over these equations. For instance, *dead-code elimination* or *code motion* are examples of applications [42]. The attractiveness of DFA is that it computes safety properties for each instruction of the program, that behave like *invariants*, i.e., they are valid for all executions.

The major problem in the reconstruction of the BBG are *indirections* of the control flow, i.e., branches whose argument is a register value rather than a assembly label [39]. In our case, the challenge is to statically determine the outcome of the branches' conditionals, that directly depends on the contents of the CPSR register. However, it should be

[2] In arithmetic, a borrow is the opposite of a carry.

noted that it might not be always possible to solve these conditionals, since, for instance, there might be a dependence with user interaction.

Static analysis allows to determine dynamic properties of programs, statically. In the following, an algorithm tailored for ARM programs that tries to cope with the conditional branches will be presented.

Let $Prog$ denote the set of all ARM programs. Further, PC is a set ranging over the natural numbers[3], and $p \in Prog$ is the program of interest, the one being analysed. The function

$$flow : Prog \rightarrow \mathscr{P}(PC_* \times PC_*),$$

maps programs to sets of flows, where PC_* is defined as a finite subset of PC, holding the program counter values of p. It should be clear that $\mathscr{P}(D)$ stands for the *powerset* of D. Formally, it is defined as

$$\mathscr{P}(D) = \{D' \mid D' \subseteq D\},$$

i.e., the set of all subsets of D, including the empty set and D itself. Hence, for the ARM program shown in listing 4.2, we obtain the following:

$$\begin{aligned} flow(p) = \quad & \{(1,2),(2,3),(3,4),(4,5),(5,6),(6,7),(7,8),(8,9), \\ & (9,10),(10,11),(11,12), \\ & (9,13),(13,14), \\ & (12,15),(14,15),(15,16)\} \end{aligned}$$

Another important characteristic of a program is the stores that it manipulates. Let us define

$$Store = Register_* \cup Mem_*,$$

as the set holding all the stores of a program, where $Register_* \subseteq Register$ is a subset of the ARM registers, and analogously for $Mem_* \subseteq Mem$ w.r.t. to the memory areas. Hence, for our running example (see listing 4.2), we got:

$$Store = \{IP, SP, FP, LR, PC, R3, R0\} \cup \{[FP, \#-16], [FP, \#-20]\}$$

[3] In the beginning of the nineteenth century a definition of the natural numbers containing the element 0 appeared. In this case, we mean the traditional definition, i.e., the positive integers.

4.1.2.1 Reaching Definition Analysis

The *Reaching Definition Analysis*[4] is a common DFA. In [42] it is defined as:

"For each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path."

While it is clear what an *assignment* means for high-level programming languages, let us clarify this concept for the ARM programming language. In the following, an assignment will be considered whenever a store is written to. For instance, the instruction *CMP R3, #6* is an assignment to the CPSR register.

The *Reaching Definition Analysis* is specified by two functions. The first,

$$kill_{rd} : Instruction \rightarrow \mathscr{P}(Store \times PC_*^?),$$

produces the set of pairs of stores and program counters of assignments that are *destroyed* by the instruction. An assignment is destroyed if the instruction assigns a new value to the store. To cope with uninitialized stores, we use the special program counter "?", hence

$$PC_*^? = PC_* \cup \{?\}.$$

Further, $Instruction$ is the set holding the ARM instructions. The definition of the $kill_{rd}$ function is as represented in Table 4.1.

The second function specifying the *Reaching Definition Analysis*,

$$gen_{rd} : Instruction \rightarrow \mathscr{P}(Store \times PC_*^?),$$

produces the set of pairs of stores and program counters of assignments generated by the instruction. Its definition can be seen in Table 4.2. Furthermore, as mentioned in section 2.2, in ARM any instruction can be suffixed by *S*. Whenever this is the case, the CPSR register needs to be updated, i.e., it is also assigned. Thus,

$$gen_{rd}(< Op > S) \quad = \quad gen_{rd}(Op) \cup \{(CPSR, pc) \mid pc \text{ is the instruction's program counter } \}$$

For our running example (see listing 4.2), the results of application of the functions $kill_{rd}$ and gen_{rd} are as illustrated by Tables 4.3 and 4.4, respectively.

$kill_{rd}$: **killed assignments**		
$kill_{rd}(BX\ R_i)$	=	$\{(R15,?)\} \cup \{(R15,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(B\ exp)$	=	$\{(R15,?)\} \cup \{(R15,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(BL\ R_i)$	=	$\{(R14,?),(R15,?)\}$
		$\cup\{(R14,?),(R15,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(MOV\ R_i, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(MVN\ R_i, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(CMP\ R_i, Op2)$	=	$\{(CPSR,?)\} \cup \{(CPSR,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(CMN\ R_i, Op2)$	=	$\{(CPSR,?)\} \cup \{(CPSR,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(TEQ\ R_i, Op2)$	=	$\{(CPSR,?)\} \cup \{(CPSR,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(TST\ R_i, Op2)$	=	$\{(CPSR,?)\} \cup \{(CPSR,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(AND\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(EOR\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(SUB\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(RSB\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(ADD\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(ADC\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(SBC\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(RSC\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(ORR\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(BIC\ R_i, R_j, Op2)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(MRS\ R_i, CPSR)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(MSR\ R_i, CPSR)$	=	$\{(CPSR,?)\} \cup \{(CPSR,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(MUL\ R_i, R_j, R_k)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(MLA\ R_i, R_j, R_k, R_l)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(LDR\ R_i, Addr)$	=	$\{(R_i,?)\} \cup \{(R_i,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(STR\ R_i, Addr)$	=	$\{(Addr,?)\} \cup \{(Addr,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(LDM\ R_i, R_{List})$	=	$\bigcup_{r \in R_{List}} \{(r,?)\} \cup \{(r,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(STM\ R_i, R_{List})$	=	$\emptyset$
$kill_{rd}(SWP\ R_i, R_j, R_k)$	=	$\{(R_i,?),(R_k,?)\}$
		$\cup\{(R_i,pc),(R_k,pc) \mid pc \text{ is pointing to an assignment}\}$
$kill_{rd}(SWI\ R_i, exp)$	=	$\emptyset$

Table 4.1: $kill_{rd}$ function

gen_{rd} : **generated assignments**		
$gen_{rd}(BX\,R_i)$	=	$\{(R15, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(B\,exp)$	=	$\{(R15, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(BL\,R_i)$	=	$\{(R14, pc), (R15, pc) \mid$ pc is the instruction's program counter $\}$
$gen_{rd}(MOV\,R_i, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(MVN\,R_i, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(CMP\,R_i, Op2)$	=	$\{(CPSR, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(CMN\,R_i, Op2)$	=	$\{(CPSR, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(TEQ\,R_i, Op2)$	=	$\{(CPSR, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(TST\,R_i, Op2)$	=	$\{(CPSR, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(AND\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(EOR\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(SUB\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(RSB\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(ADD\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(ADC\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(SBC\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(RSC\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(ORR\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(BIC\,R_i, R_j, Op2)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(MRS\,R_i, CPSR)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(MSR\,R_i, CPSR)$	=	$\{(CPSR, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(MUL\,R_i, R_j, R_k)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(MLA\,R_i, R_j, R_k, R_l)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(LDR\,R_i, Addr)$	=	$\{(R_i, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(STR\,R_i, Addr)$	=	$\{(Addr, pc) \mid pc$ is the instruction's program counter $\}$
$gen_{rd}(LDM\,R_i, R_{List})$	=	$\bigcup_{r \in R_{List}} \{(r, pc) \mid$ pc is the instruction's program counter $\}$
$gen_{rd}(STM\,R_i, R_{List})$	=	$\emptyset$
$gen_{rd}(SWP\,R_i, R_j, R_k)$	=	$\{(R_i, pc), (R_k, pc) \mid$ pc is the instruction's program counter $\}$
$genl_{rd}(SWI\,R_i, exp)$	=	$\emptyset$

Table 4.2: gen_{rd} function

killed assignments of ARM program in listing 4.2		
$kill_{rd}$(*MOV IP, SP*)	=	$\{(IP,?),(IP,0),(IP,1),(IP,2)\}$
$kill_{rd}$(*STMFD SP!, {FP, IP, LR, PC}*)	=	$\emptyset$
$kill_{rd}$(*SUB FP, IP, #4*)	=	$\{(FP,?),(FP,1),(FP,2),(FP,15)\}$
$kill_{rd}$(*SUB SP, SP, #8*)	=	$\{(SP,?),(SP,0),(SP,3),(SP,3),(SP,15)\}$
$kill_{rd}$(*MOV R3, #5*)	=	$\{(R3,?),(R3,4),(R3,5),(R3,6),(R3,7),(R3,9),$ $(R3,10),(R3,12),(R3,13)\}$
$kill_{rd}$(*STR R3, [FP, #-16]*)	=	$\{([FP,\#-16],?),([FP,\#-16],5),([FP,\#-16],6)\}$
$kill_{rd}$(*LDR R3, [FP, #-16]*)	=	$\{(R3,?),(R3,4),(R3,5),(R3,6),(R3,7),(R3,9),$ $(R3,10),(R3,12),(R3,13)\}$
$kill_{rd}$(*CMP R3, #6*)	=	$\{(CPSR,?),(CPSR,7),(CPSR,8)\}$
$kill_{rd}$(*BLE .L2*)	=	$\{(R15,?),(R15,8),(R15,11)\}$
$kill_{rd}$(*MOV R3, #1*)	=	$\{(R3,?),(R3,4),(R3,5),(R3,6),(R3,7),(R3,9),$ $(R3,10),(R3,12),(R3,13)\}$
$kill_{rd}$(*STR R3, [FP, #-20]*)	=	$\{([FP,\#-20],?),([FP,\#-20],10),([FP,\#-20],13),$ $([FP,\#-20],14)\}$
$kill_{rd}$(*B .L1*)	=	$\{(R15,?),(R15,8),(R15,11)\}$
$kill_{rd}$(*MOV R3, #0*)	=	$\{(R3,?),(R3,4),(R3,5),(R3,6),(R3,7),(R3,9),$ $(R3,10),(R3,12),(R3,13)\}$
$kill_{rd}$(*STR R3, [FP, #-20]*)	=	$\{([FP,\#-20],?),([FP,\#-20],10),([FP,\#-20],13),$ $([FP,\#-20],14)\}$
$kill_{rd}$(*LDR R0, [FP, #-20]*)	=	$\{(R0,?),(R0,14)\}$
$kill_{rd}$(*LDMEA FP, { FP, IP, PC }*)	=	$\{(PC,?)(PC,1),(PC,15),(SP,?)(SP,0),(SP,3),$ $(SP,15),(FP,?),(FP,1),(FP,2),(FP,15)\}$

Table 4.3: $kill_{rd}$ applied to the ARM program in listing 4.2

generated assignments of ARM program in listing 4.2		
gen_{rd}(*MOV IP, SP*)	=	$\{(IP,0)\}$
gen_{rd}(*STMFD SP!, {FP, IP, LR, PC}*)	=	$\emptyset$
gen_{rd}(*SUB FP, IP, #4*)	=	$\{(FP,2)\}$
gen_{rd}(*SUB SP, SP, #8*)	=	$\{(SP,3)\}$
gen_{rd}(*MOV R3, #5*)	=	$\{(R3,4)\}$
gen_{rd}(*STR R3, [FP, #-16]*)	=	$\{([FP,\#-16],5)\}$
gen_{rd}(*LDR R3, [FP, #-16]*)	=	$\{(R3,6)\}$
gen_{rd}(*CMP R3, #6*)	=	$\{(CPSR,7)\}$
gen_{rd}(*BLE .L2*)	=	$\{(R15,8)\}$
gen_{rd}(*MOV R3, #1*)	=	$\{(R3,9)\}$
gen_{rd}(*STR R3, [FP, #-20]*)	=	$\{([FP,\#-20],10)\}$
gen_{rd}(*B .L1*)	=	$\{(R15,11)\}$
gen_{rd}(*MOV R3, #0*)	=	$\{(R3,12)\}$
gen_{rd}(*STR R3, [FP, #-20]*)	=	$\{([FP,\#-20],13)\}$
gen_{rd}(*LDR R0, [FP, #-20]*)	=	$\{(R0,14)\}$
gen_{rd}(*LDMEA FP, { FP, IP, PC }*)	=	$\{(PC,15),(SP,15),(FP,15)\}$

Table 4.4: gen_{rd} applied to the ARM program in listing 4.2

Finally, the analysis is completed by setting the *data-flow equations* composed by the pair of functions

$$RD_{entry}, RD_{exit} : PC_* \to \mathscr{P}(Store \times PC_*^?),$$

which are defined as in Table 4.5.

$RD_{entry}(pc)$	=	$\begin{cases} \{(R_i, ?) \mid R_i \in stores(p)\} & \text{if } pc = 1, \\ \bigcup\{RD_{exit}(pc') \mid (pc', pc) \in flow(p)\} & \text{otherwise,} \end{cases}$ where $stores : Prog \to \mathscr{P}(Store)$ returns the stores of the program being analysed.
$RD_{exit}(pc)$	=	$(RD_{entry}(pc) \setminus kill_{rd}(Inst^{pc})) \cup gen_{rd}(Inst^{pc})$, where $Inst^{pc}$ is the instruction pointed by the program counter pc.

Table 4.5: data-flow equations

Applying RD_{entry} and RD_{exit} to our running example (see listing 4.2) gives rise to the following data-flow equations:

$$\begin{cases} RD_{entry}(1) = \{(FP,?),(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,\#-16],?), \\ \qquad\qquad (LR,?),(IP,?),(R3,?),([FP,\#-20],?),(SP,?)\} \\ RD_{entry}(2) = RD_{exit}(1) \\ \quad\vdots \\ RD_{entry}(13) = RD_{exit}(9) \\ \quad\vdots \\ RD_{entry}(15) = RD_{exit}(14) \cup RD_{exit}(12) \\ \quad\vdots \\ RD_{exit}(1) = (RD_{entry}(1) \setminus kill_{rd}(MOV\ IP, SP)) \cup gen_{rd}(MOV\ IP, SP) \\ \quad\vdots \end{cases}$$

Computing the fixpoint using Kildall's worklist algorithm [43], the solution shown in Table 4.6 is obtained.

[4] As argued in [42], this analysis would be more properly called *reaching assignments*

pc	$RD_{entry}(pc)$	$RD_{exit}(pc)$
1	{(FP,?),(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(IP,?),(R3,?),([FP,#−20],?),(SP,?)}	{(FP,?),(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(IP,1),(R3,?),([FP,#−20],?),(SP,?)}
2	{(FP,?),(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(R3,?),([FP,#−20],?),(SP,?),(IP,1)}	{(FP,?),(PC,?)(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(R3,?),([FP,#−20],?),(SP,?),(IP,1)}
3	{(FP,?),(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(R3,?),([FP,#−20],?),(SP,?),(IP,1)}	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(R3,?),([FP,#−20],?),(SP,?),(IP,1),(FP,3)}
4	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?),(LR,?), (R3,?),([FP,#−20],?),(SP,?),(IP,1),(FP,3)}	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),(R3,−?),([FP,#−20],?),(IP,1),(FP,3),(SP,4)}
5	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?),(LR,?), (R3,?),([FP,#−20],?),(IP,1),(FP,3),(SP,4)}	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?), (LR,?),([FP,#−20],?),(IP,1),(FP,3),(SP,4),(R3,5)}
6	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−16],?),(LR,?), ([FP,#−20],?),(IP,1),(FP,3),(SP,4),(R3,5)}	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−20],?) (LR,?),(IP,1),(FP,3),(SP,4),(R3,5),([FP,#−16],6)}
7	{(PC,?),(R0,?),(R15,?),(CPSR,?),(LR,?),([FP,#−20],?), (IP,1),(FP,3),(SP,4),(R3,5),([FP,#−16],6)}	{(PC,?),(R0,?),(R15,?),(CPSR,?),([FP,#−20],?), (LR,?),(IP,1),(FP,3),(SP,4),([FP,#−16],6),(R3,7)}
8	{(PC,?),(R0,?),(R15,?),(CPSR,?),(LR,?),([FP,#−20],?) (IP,1),(FP,3),(SP,4),([FP,#−16],6),(R3,7)}	{(PC,?),(R0,?),(R15,?),(LR,?),([FP,#−20],?), (IP,1),(FP,3),(SP,4),([FP,#−16],6),(R3,7),(CPSR,8)}
9	{(PC,?),(R0,?),(R15,?),(LR,?),([FP,#−20],?) (IP,1),(FP,3),(SP,4),([FP,#−16],6),(R3,7),(CPSR,8)}	{(PC,?),(R0,?),(LR,?),([FP,#−20],?),(IP,1),(FP,3), (SP,4),([FP,#−16],6),(R3,7),(CPSR,8),(R15,9)}
10	{(PC,?),(R0,?),(LR,?),([FP,#−20],?),(IP,1), (FP,3),(SP,4),([FP,#−16],6),(R3,7),(CPSR,8),(R15,9)}	{(PC,?),(R0,?),(LR,?),([FP,#−20],?),(IP,1),(FP,3), (SP,4),([FP,#−16],6),(CPSR,8),(R15,9),(R3,10)}
11	{(PC,?),(R0,?),(LR,?),([FP,#−20],?),(IP,1),(FP,3), (SP,4),([FP,#−16],6),(CPSR,8),(R15,9),(R3,10)}	{(PC,?),(R0,?),(LR,?),(IP,1),(FP,3),(SP,4), ([FP,#−16],6),(CPSR,8),(R15,9),(R3,10),([FP,#−20],11)}
12	{(PC,?),(R0,?),(LR,?),(IP,1),(FP,3),(SP,4),(R3,10) ([FP,#−16],6),(CPSR,8),(R15,9),([FP,#−20],11)}	{(PC,?),(R0,?),(LR,?),(IP,1),(FP,3),(SP,4), ([FP,#−16],6),(CPSR,8),(R3,10),([FP,#−20],11),(R15,12)}
13	{(PC,?),(R0,?)(LR,?),([FP,#−20],?),(IP,1), (FP,3),(SP,4),([FP,#−16],6),(R3,7),(CPSR,8),(R15,9)}	{(PC,?),(R0,?),(LR,?),([FP,#−20],?),(IP,1),(FP,3) (SP,4),([FP,#−16],6),(CPSR,8),(R15,9),(R3,13)}
14	{(PC,?),(R0,?),(LR,?),([FP,#−20],?),(IP,1),(FP,3), (SP,4),([FP,#−16],6),(CPSR,8),(R15,9),(R3,13)}	{(PC,?),(R0,?),(LR,?),(IP,1),(FP,3),([FP,#−16],6), (SP,4),(CPSR,8),(R15,9),(R3,13),([FP,#−20],14)}
15	{([FP,#−20],14),(R3,13),(R15,9),(PC,?),(R0,?), (LR,?),(IP,1),(FP,3),(SP,4),([FP,#−16],6),(CPSR,8), (R3,10),([FP,#−20],11),(R15,12)}	{([FP,#−20],14),(R3,13),(R15,9),(PC,?),(LR,?),(IP,1), (SP,4),([FP,#−16],6),(CPSR,8),(R3,10), ([FP,#−20],11),(FP,3),(R15,12),(R0,15)}
16	{([FP,#−20],14),(R3,13),(R15,9),(PC,?),(LR,?),(IP,1), (FP,3),(SP,4),([FP,#−16],6),(CPSR,8),(R3,10), ([FP,#−20],11),(R15,12),(R0,15)}	{([FP,#−20],14),(R3,13),(R15,9),(LR,?),(IP,1), ([FP,#−16],6),(CPSR,8),(R3,10),([FP,#−20],11),(R15,12), (R0,15),(PC,16),(SP,16),(FP,16)}

Table 4.6: ARM program's (listing 4.2) fixpoint for our Reaching Definition Analysis

4.1.2.2 Live Stores

Further, we want to know what stores are read, but not assigned to, by an instruction. Thus, we define the function

$$gen_{ls} : Instruction \rightarrow \mathscr{P}(Store),$$

that produces this set. It is defined by Table 4.7.

Applying gen_{ls} to the instructions from our running example, we obtain the results shown in Table 4.8.

As stated in section 2.2, all ARM instruction can be conditionally executed. If an instruction is suffixed by a condition code (see Table 2.2), the CPSR register is read in order to check if the condition is met. Thus,

$$gen_{ls}(< Op > CC) = gen_{ls}(Op) \cup \{CPSR\}.$$

4.1.2.3 Putting it all together

Having the reaching definition analysis' fixpoint, that give us the assignments that may have been made at each program point, and the stores read by each instruction, we can now define chains that link program points that produce a value to the program points that use them. We shall call links that for each use of a store, associate all assignments that reach that use as *Use-Definition chains* or *ud-chains*. We will define such chains,

$$UD : Store \times PC_* \rightarrow \mathscr{P}(PC_*),$$

by:

$$UD(store, pc) = \begin{cases} \{pc' \mid (store, pc') \in RD_{entry(pc)}\} & \text{if } store \in gen_{ls}(Ins^{pc}), \\ \emptyset & \text{otherwise.} \end{cases}$$

Where Ins^{pc} is the instruction being pointed by the program counter pc. For the sake of clarity, let the us consider the following application of the UD function on our running example.

$$UD([FP, \#-20], 15) = \{14, 11\}$$

gen_{ls} : **live stores**		
$gen_{ls}(BX\ R_i)$	=	$\{R_i\}$
$gen_{ls}(B\ exp)$	=	$\emptyset$
$gen_{ls}(BL\ R_i)$	=	$\emptyset$
$gen_{ls}(MOV\ R_i, Op2)$	=	$\{R_j \mid R_j \in Op2\}$
$gen_{ls}(MVN\ R_i, Op2)$	=	$\{R_j \mid R_j \in Op2\}$
$gen_{ls}(CMP\ R_i, Op2)$	=	$\{R_i\} \cup \{R_j \mid R_j \in Op2\}$
$gen_{ls}(CMN\ R_i, Op2)$	=	$\{R_i\} \cup \{R_j \mid R_j \in Op2\}$
$gen_{ls}(TEQ\ R_i, Op2)$	=	$\{R_i\} \cup \{R_j \mid R_j \in Op2\}$
$gen_{ls}(TST\ R_i, Op2)$	=	$\{R_i\} \cup \{R_j \mid R_j \in Op2\}$
$gen_{ls}(AND\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(EOR\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(SUB\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(RSB\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(ADD\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(ADC\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(SBC\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(RSC\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(ORR\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(BIC\ R_i, R_j, Op2)$	=	$\{R_j\} \cup \{R_k \mid R_k \in Op2\}$
$gen_{ls}(MRS\ R_i, CPSR)$	=	$\{CPSR\}$
$gen_{ls}(MSR\ R_i, CPSR)$	=	$\{R_i\}$
$gen_{ls}(MUL\ R_i, R_j, R_k)$	=	$\{R_j, R_k\}$
$gen_{ls}(MLA\ R_i, R_j, R_k, R_l)$	=	$\{R_j, R_k, R_l\}$
$gen_{ls}(LDR\ R_i, Addr)$	=	$\{Addr\}$
$gen_{ls}(STR\ R_i, Addr)$	=	$\{R_i\}$
$gen_{ls}(LDM\ R_i, R_{List})$	=	$\emptyset$
$gen_{ls}(STM\ R_i, R_{List})$	=	$\{R_i\}$
$gen_{ls}(SWP\ R_i, R_j, R_k)$	=	$\{Rj\}$
$gen_{ls}(SWI\ R_i, exp)$	=	$\emptyset$

Table 4.7: gen_{ls} function

stores read of ARM program in listing 4.2		
gen_{ls}(*MOV IP, SP*)	=	$\{SP\}$
gen_{ls}(*STMFD SP!, { FP, IP, LR, PC }*)	=	$\{PC, LR, IP, FP,\}$
gen_{ls}(*SUB FP, IP, #4*)	=	$\{IP\}$
gen_{ls}(*SUB SP, SP, #8*)	=	$\{SP\}$
gen_{ls}(*MOV R3, #5*)	=	$\emptyset$
gen_{ls}(*STR R3, [FP, #-16]*)	=	$\{R3\}$
gen_{ls}(*LDR R3, [FP, #-16]*)	=	$\{[FP, \# - 16]\}$
gen_{ls}(*CMP R3, #6*)	=	$\{R3\}$
gen_{ls}(*BLE. L2*)	=	$\{CPSR\}$
gen_{ls}(*MOV R3, #1*)	=	$\emptyset$
gen_{ls}(*STR R3,[FP,#-20]*)	=	$\{R3\}$
gen_{ls}(*B .L1*)	=	$\emptyset$
gen_{ls}(*MOV R3, #0*)	=	$\emptyset$
gen_{ls}(*STR R3, [FP, #-20]*)	=	$\{R3\}$
gen_{ls}(*LDR R0, [FP, #-20]*)	=	$\{[FP, \# - 20]\}$
gen_{ls}(*LDMEA FP, { FP, IP, PC}*)	=	$\emptyset$

Table 4.8: gen_{ls} applied to the ARM program in listing 4.2

This tell us that the value of the $[FP,\#-20]$ store, at program counter $pc = 15$, either comes from the instruction being pointed by the program counter $pc = 14$, or the one at $pc = 11$ (see Figure 4.1).

Typically, these *ud-chains* are part of a means for *dead code elimination* and *code motion*. In our case we are concerned with the evaluation of conditional branches. Determining the origin(s) of the value held by the CPSR register at a conditional branch instruction, will yield the instructions that influence its contents, and thus may allow us to predict the outcome of the condition.

Computing the ud-chain at the conditional branch instruction we get

$$UD(CPSR, 9) = \{8\},$$

meaning that the value of the CPSR register depends on the instruction pointed by the program counter $pc = 8$. Now, for each of the yielded program counters, it is a matter of computing its ud-chain(s). For our running example, the following would be computed:

$$\begin{aligned} UD(R3, 8) &= \{7\}, \\ UD([FP,\#-16], 7) &= \{6\}, \\ UD(R3, 6) &= \{5\}, \\ UD(*, 5) &= \emptyset, \end{aligned}$$

i.e., the only store yielding a result different than $\emptyset$ by the instruction pointed by $pc = 8$ is $R3$. From this, we determine that the value of the $R3$ register depends on the instruction being pointed by the program counter $pc = 7$. The two subsequent ud-chains are performed analogously. Finally, the instruction pointed by program counter $pc = 5$ does not yield any value, meaning that the trace of dependences has ended. This is due to the fact that in the instruction pointed by program counter $pc = 5$, *MOV R3, #5*, the store $R3$ is directly affected by a constant.

The computed ud-chains yield one trace, $[5, 6, 7, 8]$, that tells us which are the instructions necessary to consider in order to determine the outcome of the conditional branch. In this case, as we seen above, it will always evaluate to *true*, thus, we can safely remove the edge branching to the opposite outcome. This will allow to produce a more precise timing analysis, since there is no point in considering a path that can never be taken.

4.2 Pipeline Analysis

Having generated the ICFG, it is now time to cope with the overlapping effects of the pipeline. As mentioned in section 2.3, an instruction goes through a number of pipeline stages and consumes a number of cycles when it reaches the *execute* stage. The execution of a sequence of instructions overlaps in the pipeline as far as the data dependences between instructions allow it. Depending on its initial state, the execution of a sequence of instructions may result in different *traces*, i.e., sequence of execution states, and thus different number of cycles. Naturally, the length of a trace, is the number of cycles that this execution took. Thus, *concrete execution* can be seen as the execution of a sequence of instructions, starting in a concrete pipeline state, producing a trace of such states, whose length is the number of cycles it spent.

In the following, before moving to the analysis of the pipeline itself, we will first introduce and motivate the general concepts of the abstract interpretation framework.

4.2.1 Abstract Interpretation

A program is characterized by its control-flow graph, with a set of edges $E \subseteq V \times Ins \times V$ and with the induced set of vertices, where V represents the program points, $v_i \in V$ models the program's entry point and Ins models the instruction to be executed whenever taking that edge.

A semantic function $[\![.]\!] : Ins \rightarrow (S \rightarrow S)$ assigns to each $ins \in Ins$, a transfer function that models its effect on the program state S, being evaluated. For instance, in the ARM instruction set [10], the instruction *B address* is specified as $R15 := address$, i.e., update of the program counter register to the address given by the evaluation of the expression *address*, and thus would have to be modelled accordingly to its specification.

The collecting semantics assigns for each program point V, the set of program states S, which may occur in any possible execution, i.e., $CS : V \rightarrow \mathscr{P}(S)$ (where $\mathscr{P}(S)$ stands for powerset of S). The analysis to be performed, can be specified by extracting a number of equations from the program being considered. There are two types of equations. The first one, relates exit with entry information for each program point V. While the second, relates entry information of a program point V_i, with exit information of nodes from which there exists an edge to the program point V_i, i.e., $\bigcup\{V_j \mid (V_j, ins, V_i) \in E\}$.

The resulting system of equations can be solved by computing the least fixpoint $lfp(F) =$

$F^n(\lambda v.\emptyset)$ of the functional $F : (V \to \mathscr{P}(S)) \to (V \to \mathscr{P}(S))$:

$$F(f)(v') = \begin{cases} S_0 & \text{if } v' = v_{in}, \\ \bigcup_{(v,ins,v') \in E} [\![ins]\!](f(v)) & \text{otherwise,} \end{cases}$$

where $S_0 \subseteq S$ is the set of the program's initial states.

However, as mentioned earlier computing the collecting semantics of a large and complex program P can be too expensive to be feasible. Hence, the analysis is performed on a simpler abstract domain $\mathscr{D}_\alpha = (S, L, \beta, \gamma)$, where $L = (L, \sqsubseteq, \bigsqcup, \bot, \top)$ is a complete semilattice and $\beta : S \to L$ is a representation function, mapping concrete to abstract states. The idea, is that β maps a state S to the best property describing it. Finally, $\gamma : L \to \mathscr{P}(S)$ is a concretization function mapping abstract states to concrete states.

As we seen above, the collecting semantics operates over sets of states, while our abstract domain, operates over sets of properties. Thus, with the purpose of relating these two domains, we define an abstraction function $\alpha : \mathscr{P}(S) \to L$, by $\alpha(S') = \bigsqcup\{\beta(s) \mid s \in S'\}$. The concretization function γ, and the abstraction function α, will therefore yield the following relation:

$$\mathscr{P}(S) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} L$$

If for the above relation the following condition

$$\alpha(X) \sqsubseteq l \Leftrightarrow X \subseteq \gamma(l),$$

holds, we shall call the pair (α, γ) a *Galois connection.* Furthermore, in order to ensure termination we require the *Ascending Chain Condition* to hold, i.e., every ascending chain of elements eventually terminates. For this, both the abstraction function α, and the concretization function γ, must be monotonic w.r.t. the $\sqsubseteq$ and $\subseteq$ operators, respectively.

The semantic function defined above, can now be redefined as an *abstract semantic* function $[\![.]\!]_\alpha : Ins \to (L \to L)$, over the abstract domain. The abstract counterparts of the transfer functions $[\![ins]\!]$, i.e., $[\![ins]\!]_\alpha$, must also be monotonic w.r.t. the $\sqsubseteq$ operator. Finally, the analysis can now be applied with the *abstract collecting semantics* $CS_\alpha : V \to L$, such that $\forall v \in V : CS(v) \subseteq \gamma(CS_\alpha(v))$, i.e., the computed results are precise, or an over-approximation of the collecting semantics, and thus are safe.

The resulting system of equations can be solved by computing the fixpoint $lfp(F_\alpha) = F_\alpha^n(\lambda v.\bot)$ of the functional $F_\alpha : (V \to L) \to (V \to L)$:

$$F_\alpha(f)(v') = \begin{cases} l_0 & \text{if } v' = v_{in}, \\ \bigsqcup_{(v,ins,v') \in E} [\![ins]\!]_\alpha(f(v)) & \text{otherwise,} \end{cases}$$

where the abstraction of the *concrete* initial states is defined as initial abstract state, thus $\alpha(S_0) \sqsubseteq l_0$.

It should be clear that, since the abstract transfer functions, $[\![ins]\!]_\alpha$, are monotonic w.r.t. the $\sqsubseteq$ operator, by induction we obtain $F_\alpha^n(\lambda v.\bot) \sqsubseteq F_\alpha^{n+1}(\lambda v.\bot)$ for all n. All the elements of the sequence are in L, and since this is a finite set, not all elements of the sequence can be distinct, thus, there must be some n such that:

$$F_\alpha^{n+1}(\lambda v.\bot) = F_\alpha^n(\lambda v.\bot)$$

Furthermore, since $F_\alpha^{n+1}(\lambda v.\bot) = F_\alpha(F_\alpha^n(\lambda v.\bot))$, we have reached the least fixpoint of F_α, i.e., $lfp(F_\alpha)$, and thus a solution to the equation system.

The analysis to be instantiated depends on the target processor being evaluated. In our current prototype implementation of ACCEPT, we focus on the ARM7TDMI-S processor (see Chapter 2). Featuring a three-stage pipeline, the analysis to be performed must cope with the overlapping effects of instruction pipelining.

4.2.2 Analysis

Pipeline analysis tries to find how instructions move through the pipeline. However, rather than considering concrete execution on a *concrete pipeline model*[5], it regards *abstract execution* of sequences of instructions on a *abstract pipeline model* [6]. This execution produces *abstract traces*, i.e., sequences of *abstract states*, where information contained in the concrete states might be missing. Register contents are unknown, thus, w.r.t. WCET bound prediction, in order to obtain a safe analysis the worst-case scenario must always be assumed to happen. For instance, the *MUL* instruction takes $m + 1$ cycles to execute, the value of m depending on the contents of the operand, since information about register contents is lacking, for a safe upper-bound m must be assumed to have value 4 (see Table 2.3).

[5]Which may be described in some formal language such as VHDL.

[6]This abstract model is obtained by eliminating the components that are not relevant for the timing behaviour, and can also be described in a language like VHDL.

In the following, we will present our firsts steps towards the analysis of the pipeline. We base ourselves on the approach for pipeline analysis of a state-of-the-art processor described in [16].

In [16], the algorithm for pipeline analysis feeds to the abstracted pipeline model the instructions from a *basic block*, starting in the abstract pipeline state, and producing an abstract trace. Further, since the considered processor possesses a cache, and there might be basic blocks with more than one predecessor, one analysis for each pair of basic block and context per initial state is performed. From the set of yielded traces, the one with greatest length is taken as the upper-bound for this basic block in this context. In our case however, we are also concerned with the BCET, but only need to cope with the effects of the pipeline.

Hence, an abstract state is a 3-tuple, where the first component is a set of states of the timing model $s_\alpha \in S_\alpha$(like in [16]), the second component is a bound on the BCET and the third component a bound on the WCET. For the set S_α, let $\mathscr{P}(S_\alpha) \times Z \times Z$ be ordered by

$$(S_{\alpha 1}, b_1, w_1) \sqsubseteq (S_{\alpha 2}, b_2, w_2) \text{ iff } S_{\alpha 1} \subseteq S_{\alpha 2} \wedge b_1 \geq b_2 \wedge w_1 \leq w_2$$

Our *join* operator will then be defined by

$$\bigsqcup\{(S_i, b_i, w_i \mid i = 1, ..., k)\} = (\bigsqcup\{S_i \mid i = 1, ..., k\}, \min_i b_i, \max_i w_i)$$

An abstract timing model is a state machine whose transitions corresponds to clock cycles. In our case we have to consider the three-stages of the pipeline, and keep track on which stage the instructions is on. In [16] a fictitious stage is added in order to determine whenever an instruction has finished execution. Further, no Galois Connections was established. Instead, they rely solely on a concretization function.

Indeed, up to the present, there is no suitable representation of sets of concrete pipeline states to single abstract states, and is very likely hard to find [12].

4.3 Bound Calculation

The previous pipeline analysis determined execution time bounds on the basic blocks. Determining the WCET and BCET is now a matter of finding the paths that lead to the

longest and shortest execution time, respectively. In timing analysis this is usually achieved by means of Integer Linear Programming (ILP) techniques.

Let t_{wcet} denote the longest execution time of a program. Computing t_{wcet} could be performed simply by summing the execution time of the n nodes $v \in V$, belonging to the path leading to the longest execution time:

$$t_{wcet} = \sum_{i=1}^{n} t(v_i),$$

where $t(v_i)$ is the execution time of the node v_i. This could be feasible, if this path leading to the longest execution would be known beforehand. Determining it can be too much expensive for real programs. To surpass this obstacle, rather than analysing paths explicitly, Implicit Path Enumeration (IPE) counts the number of executions of basic blocks. Hence, the above sum can be reformulated as:

$$t_{wcet} = \sum_{v \in V} t(v_i).cnt(v_i),$$

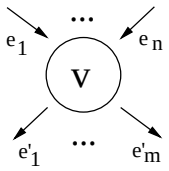
where $cnt(v_i)$ stands for the execution count of the node v_i. This sum is a linear combination, since the execution times $t(v_i)$ are constants, as they have been computed in the previous pipeline analysis. Moreover, it is clear that the $cnt(v_i)$'s must be integer values. These are constrained according to the program's behaviour. There are two types of constraints: *program structural constraints* and *program functionality constraints*. The former constraints are derived automatically from the program's BBG, while the latter are either derived via a previous static analysis or (preferably) provided by the user, for instance, via annotations.

The number of times that the execution reaches a node v, is equal to the number of times that the execution leaves the node, which is also equal to the node execution count, i.e. $cnt(v)$. This is depicted in Figure 4.2. This principle can be applied as a basis to automatically obtain the structural constraints.

However, the structural constraints do not tell anything about loop bounds. This information is supplied as functionality constraints, and since they cannot be always automatically inferred, they are therefore usually provided as annotations by the user.

Let us demonstrate this by means of an example retrieved from [45] . Figure 4.3 depicts a basic block graph of a program with an if-statement nested within a loop.

Using the principle depicted in Figure 4.2, the structural constraints are inferred from the basic block graph, yielding:



$$\sum_{i=1}^{n} cr(e_i) = cnt(v) = \sum_{i=1}^{m} cr(e'_i)$$

Figure 4.2: The number of times that the incoming and outgoing edges are crossed is equal, and so is the execution count of v

The first structural constraint stipulates that the edge e_1 is crossed once. The remaining are inferred by applying the principle shown in Figure 4.2. For instance, looking on the third structural constraint, the execution count of the basic block represented by node v_2, is equal to the number of times that its incoming edges, e_2 and e_8, are crossed, and also equal to the number of times that its outgoing edges, e_3 and e_9, are crossed.

However, as stated above, the structural constraints do not express any information regarding loop bounds. The idea is that the user may have domain-specific knowledge (e.g. ranges for input values), and thus can leverage it as functionality constraints. For instance, by knowing that the value of the variable k, is positive before entering the loop, we know that the loop body will be executed between 0 and 10 times. Thus, the following functionality constraint can be defined:

$$0cnt(v_1) \leq cnt(v_3) \leq 10cnt(v_1)$$

Besides loop bounds, functionality constraints can also be used to cope with other aspects of the program's behaviour. A closer inspection of the code shows that node v_5 can only be executed once. In order to deal with this, we define another functionality constraint:

$$\begin{aligned}
cr(e_1) &= 1 \\
cnt(v_1) &= cr(e_1) = cr(e_2) \\
cnt(v_2) &= cr(e_2) + cr(e_8) = cr(e_3) + cr(e_9) \\
cnt(v_3) &= cr(e_3) = cr(e_4) + cr(e_5) \\
cnt(v_4) &= cr(e_4) = cr(e_6) \\
cnt(v_5) &= cr(e_5) = cr(e_7) \\
cnt(v_6) &= cr(e_6) + cr(e_7) = cr(e_8) \\
cnt(v_7) &= cr(e_9) = cr(e_{10})
\end{aligned}$$

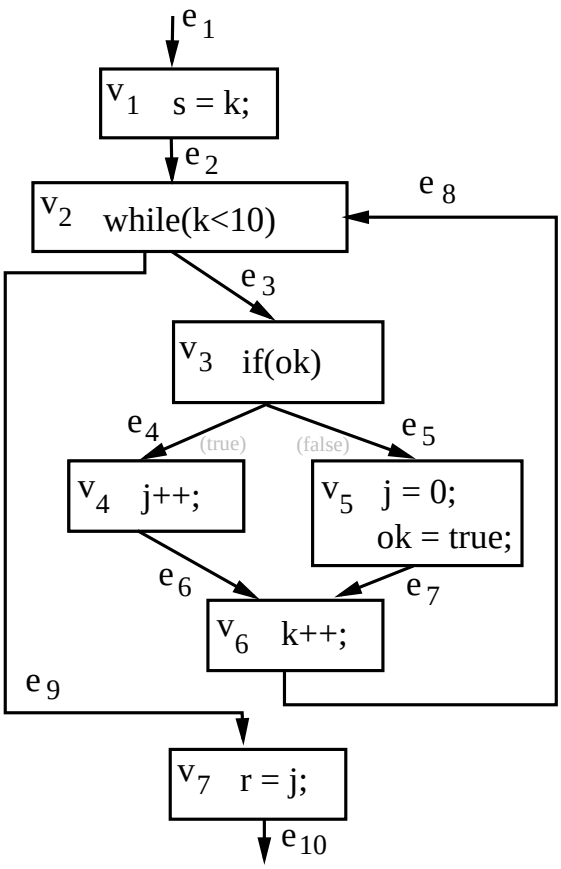


Figure 4.3: Basic block graph

$$cnt(v_5) \leq 1 cnt(v_1)$$

Determining the WCET of the BBG in Figure 4.3 is now performed using t_{wcet}'s sum in the objective function of the ILP and maximizing it accordingly with the defined structural and functionality constraints. Therefore, the objective function of the generated ILP is as follows.

$$max: \sum_{v \in V} t(v_i).cnt(v_i),$$

It should be clear that determining the BCET is performed analogously. Rather than maximizing the objective function, it is minimized.

4.4 Certificate Production

In section 2.3 a pipeline analysis was performed, by which bounds on the execution times of the basic blocks were produced. This was achieved by means of a fixpoint computation. This fixpoint allowed us to calculate both the BCET and WCET by means of ILP techniques (see section 4.3), and with this verify the compliance with a giving timing specification. However, we can also let this fixpoint play the role of a certificate[7].

In the context of mobile code safety one cannot trust the origin of the program. Hence, by adding to the code the certificate and sending both to a program consumer, it can be performed a local and independent check of the program's timing behaviour, thereby avoiding the need to trust in the code producer.

4.5 Certificate Validation

The program consumer receives a program with its certificate. In order to check the compliance with the intended timing specification, the first step is to compute the program's ICFG and verify that the certificate is a valid *abstraction*. Then, since the certificate is supposedly a fixpoint, the checking procedure can be written as:

$$Check(Certificate) = \begin{cases} True & \text{if } F_\alpha(Certificate) = Certificate, \\ False & \text{otherwise.} \end{cases}$$

Since the certificate is supposed to be a fixpoint, another iteration over it cannot change anything, thus, on the program consumer side, a simple one-pass computation is sufficient to check that the certificate is indeed a fixpoint.

In case that the received certificate do not behave as a fixpoint, the program consumer can simply reject the program. One could argue that we could let the program run, and *kill* its execution in the case of a timing behaviour non-compliance. However, that would be a waste of resources, and in the context of embedded systems, that tend to have very limited computational resources, is unacceptable. On the other hand, if the certificate is indeed a fixpoint, then the program consumer can locally compute the BCET and WCET by the ILP techniques described in section 4.3, and thus check the compliance with the

[7] Since the certificate is in fact a fixpoint, Abstraction-Carrying Code is sometimes also referred as *Fixpoint-Carrying Code* [46]

timing specification. Furthermore, it should be noted that in this framework, it is also possible for the program consumer to define new timing policies. For instance, one can be interested in tightening the timing constraints.

This validation process requires that both the producer and consumer share the same abstract transfer functions. Indeed, if the consumer used different abstract transfer functions the certificate checking process would be inefficient, and thus prohibitive for such scarce resource equipments as embedded systems. One could argue that the independence in the timing validation process is compromised by that fact, however, it should be noted that the trusted computing base is limited to this checking operation, i.e., a simple, easy-to-trust fixpoint checker, that only has to perform a one iteration process. Hence, this approach allows to detect if a program has been tampered with, since an adulteration in the program code would be detected when performing the checking operation, i.e., the fixpoint iteration. In the context of mobile code, this is particularly relevant since, rather than simply put a blind confidence on a previous timing analysis, one can validate the program's timing behaviour by solely relying on a fixpoint checker.

In this chapter we presented the fundamentals that underlies the tool developed in the context of this thesis. In the next chapter we present CATA - CertificAtes for Timing Analysis.

Chapter 5

CATA - CertificAtes for Timing Analysis

In this chapter we present an overview of the functionalities of CATA, the tool that implements the fundamentals presented in the previous chapter.

5.1 Main Features

CATA is the tool developed in the context of this work. It is a command-line application, written in OCaml [47], that features the options displayed in Figure 5.1.

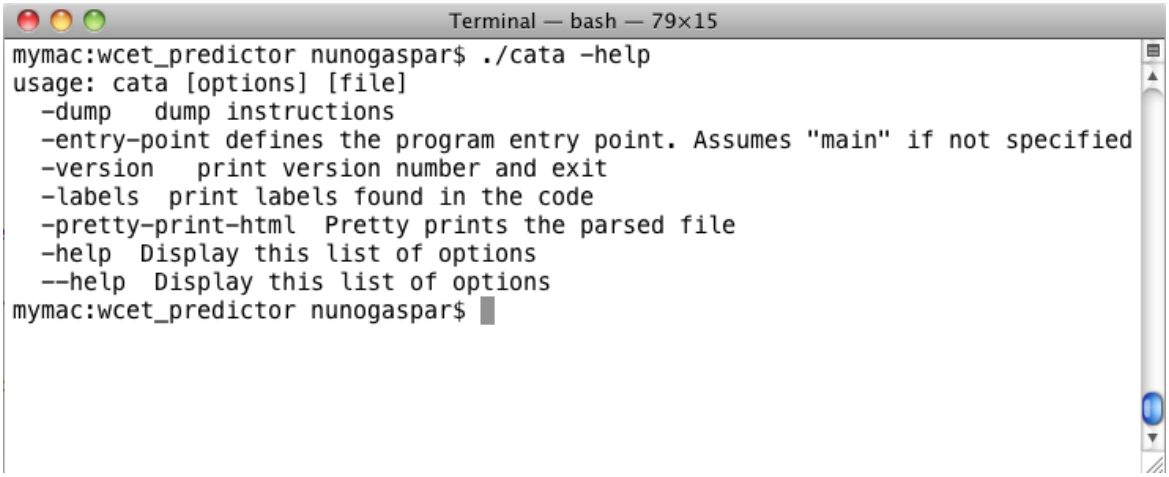


Figure 5.1: CATA - CertificAtes for Timing Analysis

It was developed as part of the ACCEPT project, with the goal of computing timing bounds and producing a certificate that could be used to perform an independent validation w.r.t the predicted bounds. However, CATA can also be used as a standalone appli-

cation, provided that the input is either an ARM assembly file or in ARTIST 2 Interchange Format (AIR) format. The architecture of CATA is depicted in Figure 5.2.

Analysis input.

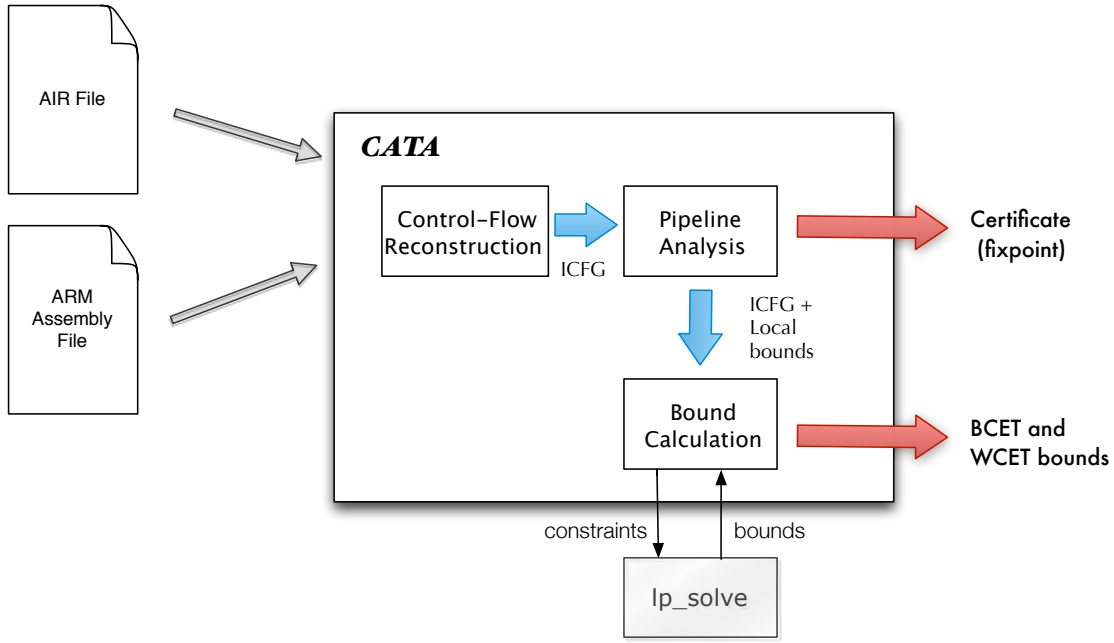


Figure 5.2: CATA's architecture

The analysis input can be either an ARM assembly file or an AIR file [48]. AIR files are used to represent control-flow graphs for real-time systems analyses. This format is based on Common Representation Language 2 (CRL2), a format developed in cooperation by Saarland University and AbsInt Angewandte Informatik GmbH, and is used as an internal intermediary representation for the tools developed at AbsInt (e.g. aiT).

The goal of AIR is to allow different timing analysis tools to exchange code and analysis results. Its development was made together with the cooperation of world-leading timing analysers tool providers, within the ARTIST2 project [49]. This goal originated several thesis [50, 51, 52], all with the objective of fostering collaboration between different tools. For instance, in [50] the purpose of this MSc thesis was to adapt the SWEET timing analyser to cope with the AIR format so that it could interact with aiT.

In order to promote cooperation with other timing analysers tools, CATA supports the AIR format, both as an input for timing analysis, and as output when receiving an ARM assembly file as input. However, the only available documentation regarding the AIR

format [48], dates from 2006, is labelled as ongoing work, and lacks information about its grammar definition. Nevertheless, our implementation of the AIR format is still strongly based on the one found in [48], and is discussed in Appendix 6.

As shown in Figure 5.2, there is an external element involved in the bound computations: *lp_solve* [53]. *lp_solve* is an open-source ILP solver that is used by CATA to compute bounds on the BCET and WCET.

The process of checking the generated certificate is performed by *CATA_check*, a tiny version of CATA. It starts by extracting the ICFG of the received program and confronts it with the received certificate. Then, performing one iteration over it will yield the validity of the program, and allow the computation of the BCET and WCET.

Chapter 6

Conclusions and Future Work

Abstract Interpretation has been widely used in the industry, being static timing analysis one of its most successful applications [12]. In our approach we also plan on using the Abstract Interpretation framework as the underlying technique, enabling us to take into account the effects of instruction overlapping caused by the pipeline. By explicitly following a standard fixpoint computation strategy [43], we are then able to apply standard integer linear programming techniques that yield the BCET and WCET.

Moreover, this fixpoint computation will allow us to infer an abstract model of the program, which can then be used as a certificate, i.e., a program consumer can locally validate the received program w.r.t. to its timing behaviour, by simply checking that this abstract model is indeed a fixpoint (a one-pass process), and then compute the BCET and WCET with the received certificate.

Our current prototype implementation of CATA is still a work in progress. At this stage there are still some open issues that remain to be addressed. Indeed, the all idea behind ACCEPT should be seen as a proof-of-concept with still large room for improvement. Namely:

- Rather than a binary file, CATA considers an ARM assembly file as input. In the context of ACCEPT, since it features a WCET-aware compilation process, considering receiving an ARM assembly file as input is perfectly feasible, in view of the fact that the timing analysis would be integrated with the compilation process itself. However, as a standalone application, CATA would need to cope with binary files in order to make it suitable for a real scenario.

  While it might seem that a decompilation process from a binary would be straight-

forward, there are in fact several challenges associated with it [54]. For instance, doing a simple direct mapping between bit-patterns and instructions would be error-prone, since some compilers insert switch tables between instructions. Furthermore, there is actually a decompilation framework based on abstract interpretation [55].

- The data-flow analysis for conditional branches presented in Section 4.1 is limited to a intraprocedure analysis. Extending it to a interprocedural analysis would be of high benefit to enhance the power of the analysis. Also, this analysis is only applied to branch instructions, but indeed, since in ARM all instructions can be conditionally executed, the same principle could be applied to the remaining instructions. For instance, detecting that a *MOVLT* instruction would never be executed, would allow us to determine that it would only consume 1 cycle (see Table 2.3), and thus leverage a more accurate timing analysis.

- One of the main challenges that we face in order to make ACCEPT useful in practice is the size of the produced certificates. Embedded systems are known for their scarce resources, and thus, cannot afford to waste computational means. In [56], Albert et al introduce the notion of a *reduced certificate*, with the objective of producing a certificate that only contains the essential information which the program consumer cannot reproduce by itself, while not yielding an overhead in the certificate checking process. Moreover, in [57] a certified fixpoint compression algorithm is presented that is able to produce reduced certificates from the results of *untrusted* static analysers.

- Our pipeline analysis is still not completed. For now, we let the fixpoint computed in the data-flow analysis for conditional branches play the role of the certificate. The calculated execution times do not take into account the overlapping effects of the pipeline. Hence, although these computed bounds are safe, they are not as precise as they would be if the pipeling effects were taken into account.

Our actual focus on the certification generation part of the *ACCEPT* platform is now on the pragmatical evaluation of our proposal. For now we are not concerned with performance, but with correctness and adequacy (in the context of mobile code). However, a timing analysis platform is only useful if it provides tight and safe bounds. In this sense, we use state of the art algorithms for their calculation. Nevertheless, comparing

equitably this kind of platform against reference tools [7], even pragmatically in the form of benchmark, is not a trivial task, the same source-code, compilation process and/or low-level code and target architecture must be considered. However, as future work we also plan to report on case studies and practical results of our framework.

To the best of our knowledge this is the first work applying the concepts of Abstraction-Carrying Code to the static timing analysis field.

References

[1] Greger Ottosson and Mikael Sjodin. Worst case execution time analysis for modern hardware architectures. In *In: Proc. of ACM SIGPLAN, Workshop on Languages, Compilers and Tools for Real-Time Systems*, pages 47–55, 1997.

[2] Jan Reineke, Bjorn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th Intl Workshop on WCET Analysis*, 2006.

[3] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[4] Mads Christian Olesen Andreas Engelbredt Dalsgaard and Martin Toft. Modular Execution Time Analysis using Model Checking (METAMOC). Master's thesis, Aalborg University, Denmark, June 2009.

[5] Karl Lermer, Colin J. Fidge, and Ian J. Hayes. A theory for execution-time derivation in real-time programs. *Theor. Comput. Sci.*, 346(1):3–27, 2005.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, New York, NY, USA, 1977. ACM.

[7] Christian Ferdinand and Reinhold Heckmann. ait: worst case execution time prediction by static program analysis. In *IFIP Congress Topical Sessions*, pages 377–384, 2004.

[8] Alexander D. Stoyen and Plamen V. Petrov. Towards a mobile code management environment for complex,real-time, distributed systems. *Real-Time Syst.*, 21(1/2):165–189, 2001.

[9] Christoph M. Kirsch, Marco A. A. Sanvido, and Thomas A. Henzinger. A programmable microkernel for real-time systems. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 35–45, New York, NY, USA, 2005. ACM.

[10] Steve Furber. *ARM System-on-Chip Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[11] Trevor Harmon and Raymond Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *Proc. 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Washington, 2007. IEEE Computer Society.

[12] Reinhard Wilhelm and Björn Wachter. Abstract interpretation with applications to timing validation. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 22–36, Berlin, Heidelberg, 2008. Springer-Verlag.

[13] Reinhard Wilhelm. Why ai + ilp is good for wcet, but mc is not, nor ilp alone. In *In Verification, Model Checking and Abstract Interpretation (VMCAI), LNCS 2937*, 2004.

[14] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[15] Manuel V. Hermenegildo, Elvira Albert, Pedro López-García, and Germán Puebla. Abstraction carrying code and resource-awareness. In *PPDP*, pages 1–11, 2005.

[16] Stephan Thesing. *Safe and Precise WCET Determination by Abstraction Interpretation of Pipeline Models*. PhD thesis, Saarland University, Germany, 2004.

[17] Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In Jens Palsberg and Zhendong Su, editors, *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *LNCS*, pages 120–136. Springer-Verlag, August 2009.

[18] Stephen Wolfram. *A New Kind of Science*. Wolfram Media, January 2002.

[19] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[20] Reinhold Heckmann, 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):1038–1054, July 2003.

[21] RapiTime. Rapitime white paper: Worst-case execution time analysis. `www.rapitasystems.com`, 2007.

[22] Guillem Bernat, Antoine Colin, and Stefan Petters. pwcet: a tool for probabilistic worst-case execution time analysis of real-time systems. January 2003.

[23] SymTA/P. Symta/p tool of tu braunschweig. `http://www.ida.ing.tu-bs.de/forschung/publikationen/symbolische_laufzeitanalyse_fuer_prozesse_symtap/`, 2009.

[24] Thomas Lundqvist. A wcet analysis method for pipelined microprocessors with cache memories. PhD thesis, Dept. of Computer Engineering, Chalmers University of Technology, 2002.

[25] Andreas Ermedahl Jan Gustafsson. Sweet - swedish execution time tool. `http://www.mrtc.mdh.se/projects/wcet/sweet.html`, 2006.

[26] Isabelle Puaut. Heptane. `http://www.irisa.fr/alf/index.php?option=com_content&view=article&id=29&Itemid=&lang=fr`, 2010.

[27] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos. `http://www.comp.nus.edu.sg/~rpembed/chronos/`, 2007.

[28] Adrian Prantl, Markus Schordan, and Jens Knoop. Tubound: A conceptually new tool for worst-case execution time analysis, 2008.

[29] Peter Puschner. Tu-vienna. `http://www.wcet.at/`, 2004.

[30] Andreas Ermedahl. *A Modular Tool Architecture for Worst-Case ExecutionTime Analysis*. VDM Verlag, Saarbrücken, Germany, Germany, 2008.

[31] Björn Lisper. Fully automatic, parametric worst-case execution time analysis. In *In Workshop on Worst-Case Execution Time (WCET) Analysis*, pages 3–0531, 2003.

[32] Karl Crary and Stephnie Weirich. Resource bound certification. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198, New York, NY, USA, 2000. ACM.

[33] Armelle Bonenfant, Christian Ferdin, Kevin Hammond, and Reinhold Heckmann. Worst-case execution times for a purely functional language. In *In 18th IFL 2006*. Springer, 2007.

[34] Steffen Jost, Kevin Hammond, Hans-Wolfgang Loidl, and Martin Hofmann. Static determination of quantitative resource usage for higher-order programs. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 223–236, New York, NY, USA, 2010. ACM.

[35] Gilles Barthe, Lennart Beringer, Pierre Crégut, Benjamin Grégoire, Martin Hofmann, Peter Müller, Erik Poll, Germán Puebla, Ian Stark, and Eric Vétillard. MOBIUS: Mobility, ubiquity, security — objectives and progress report.

[36] Mobile Resource Guarantees Official Web page. `http://groups.inf.ed.ac.uk/mrg/`, 2005.

[37] J. Kwon, A. Wellings, and S. King. A safe mobile code representation and run-time architecture for high-integrity real-time java programs. In *Proceedings of the Work-In-Progress Session, 22nd IEEE Real-Time Systems Symposium, YCS 337*, pages 37–40. Department of Computer Science, University of York, 2001.

[38] Wolfram Amme. Safetsa: a type safe and referentially secure mobile-code representation based on static single assignment form. pages 137–147. ACM Press, 2001.

[39] Daniel Kästner and Stephan Wilhelm. Generic control flow reconstruction from assembly code. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 46–55, New York, NY, USA, 2002. ACM.

[40] GNU. GNU ARM toolchain for cygwin, linux and macos. `http://www.gnuarm.com/`, 2006.

[41] Andrea Pagni, Danilo Pietro Lucini, Fabrizio Pau, Antonio Maria Borneo, and Vittorio Zaccaria. Process for translating instructions for an arm-type processor into instructions for a lx-type processor; relative translator device and computer program product. http://www.patents.com/process-translating-instructions-arm-type-processor-instructions-a-html, 2007.

[42] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[43] Gary A. Kildall. A unified approach to global program optimization. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, New York, NY, USA, 1973. ACM.

[44] Reinhard Wilhelm. Determining bounds on execution times. In R. Zurawski, editor, *Handbook on Embedded Systems*, pages 14–1,14–23. CRC Press, 2005.

[45] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium*, page 254, Washington, DC, USA, 1996. IEEE Computer Society.

[46] Soonho Kong. Introduction to proof-carrying code. http://ropas.snu.ac.kr/~soon/talk/20090807.pdf, 2009.

[47] INRIA. Ocaml official website. http://caml.inria.fr/, 2010.

[48] Henrik Theiling. Air file format specification and remarks about crl2. www.absint.com/artist2/doc/crl2/air.pdf, 2006.

[49] ArtistDesign. Artistdesign european network of excellence on embedded systems design. http://www.artist-embedded.org/artist/, 2010.

[50] Per Wolde. Adapting the wcet analysis tool sweet to the air data format. http://www.idt.mdh.se/examensarbete/index.php?choice=show&lang=en&id=0536, 2007.

[51] Deepthi Devaki A.R. Writing a powerpc assembler to alf translator in cooperation with world-leading wcet tool providers. http://www.idt.mdh.se/examensarbete/index.php?choice=show&lang=en&id=0917, 2009.

[52] Nithya Vijay. Writing a necv850e assembler to alf translator in cooperation with world-leading wcet tool providers. http://www.idt.mdh.se/examensarbete/index.php?choice=show&lang=en&id=0875, 2009.

[53] Michel Berkelaar. lp_solve official web site. http://lpsolve.sourceforge.net/5.5/, 2010.

[54] Henrik Theiling. Control flow graphs for real-time system analysis: Reconstruction from binary executables and usage in ilp-based path analysis, 2002.

[55] Johannes Kinder, Helmut Veith, and Florian Zuleger. An abstract interpretation-based framework for control flow reconstruction from binaries. In Markus Müller-Olm Neil D. Jones, editor, *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2009)*, volume 5403 of *LNCS*, pages 214–228. Springer, 2009.

[56] Elvira Albert, Puri Arenas-Sánchez, Germán Puebla, and Manuel V. Hermenegildo. Reduced certificates for abstraction-carrying code. In *ICLP*, pages 163–178, 2006.

[57] Frédéric Besson, Thomas Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3):273–291, 2006.

[58] Advanced RISC Machines ARM. ARM7DTMI data sheet, 1995.

[59] Henrik Theiling, Universitat Des Saarlandes, and Absint Angewandte Informatik Gmbh. Extracting safe and precise control flow from binaries. In *In Proc. 7th Conference on Real-Time Computing Systems and Applications*, 2000.

Acronyms

RISC Reduced Instruction Set Computer

CPSR Current Program Status Register

PC Program Counter

LR Link Register

SP Stack Pointer

CFG Control-Flow Graph

ICFG Interprocedural Control-Flow Graph

CG Call Graph

BBG Basic Block Graph

RTS Real-Time Systems

WCET Worst-Case Execution Time

BCET Best-Case Execution Time

CPI Clock Cycles per Instruction

DFA Data-Flow Analysis

PCC Proof-Carrying Code

ACC Abstraction-Carrying Code

ACCEPT Abstraction-Carrying CodE Platform for Timing validation

IPE Implicit Path Enumeration

ILP Integer Linear Programming

CATA CertificAtes for Timing Analysis

AIR ARTIST 2 Interchange Format

CRL2 Common Representation Language 2

AIR Specification Format

In this Appendix we describe the grammar of the AIR format.

.1 Language Grammar

In the following listing 1, we present our implementation of the AIR format. Terminal symbols are written in capitals and their meaning should always be clear.

Listing 1: AIR Grammar

```
air:
  | EOF
  | header body END  EOF

header:
  | CRL
      SPECIFICATION specification_name
      IMPLEMENTATION implementation_name
      VERSION
         generation_code
         safety_code
         change_code
         implementation_version
         implementation_subversion

specification_name:
  | UUIDidentifier

implementation_name:
```

```
20    | UUIDidentifier
21
22  UUIDidentifier:
23    | IDENT
24
25  dummy:
26    | /*empty*/
27    | IDENT
28
29  generation_code:
30    | INT
31
32  safety_code:
33    | INT
34
35  change_code:
36    | INT
37
38  implementation_version:
39    | INT
40
41  implementation_subversion:
42    | INT
43
44  lc_name:
45    | IDENT
46
47  body:
48    | body body_element
49    | body_element
50
51  body_element:
52    | GLOBAL global
53    | ROUTINE routine
54    | DATA data
55    | META meta
```

```
56
57 lc_id_def:
58   | lc_name
59
60 global:
61   | simple_item_lc_name_and_global_special
62
63 simple_item_lc_name_and_global_special:
64   | lc_id_def global_special lc_name_attr_list COLUMN
65   | lc_id_def global_special COLUMN
66
67 lc_name_attr_list:
68   | TWODOTS attr_lc_name_comma_list
69
70 attr_lc_name_comma_list:
71   | attr_lc_name COMMA attr_lc_name_comma_list
72   | attr_lc_name
73
74 attr_lc_name:
75   | lc_name EQUALS value
76
77 global_special:
78   | dummy
79
80 routine:
81   | lc_id_def optional_lc_name_ctxt_attr_list COLUMN
82   | lc_id_def optional_lc_name_ctxt_attr_list
83       LBRAC optional_context_def_list optional_block_list RBRAC
84
85 optional_context_def_list:
86   | /*empty*/
87   | context_def_list
88
89 context_def_list:
90   | context_def context_def_list
91   | context_def
```

```
92
93 optional_block_list:
94   | /*empty*/
95   | block_list
96
97 block_list:
98   | block block_list
99   | block
100
101 optional_lc_name_ctxt_attr_list:
102   | /*empty*/
103   | lc_name_ctxt_attr_list
104
105 lc_name_ctxt_attr_list:
106   | TWODOTS attr_lc_name_ctxt_comma_list
107
108 attr_lc_name_ctxt_comma_list:
109   | lc_name_ctxt COMMA attr_lc_name_ctxt_comma_list
110   | lc_name_ctxt
111
112 block:
113   | lc_id_def block_special lc_name_ctxt_attr_list COLUMN
114   | lc_id_def block_special lc_name_ctxt_attr_list
115       LBRAC edge_list instruction_list RBRAC
116
117 edge_list:
118   | edge_list edge
119   | edge
120
121 instruction_list:
122   | instruction instruction_list
123   | instruction
124
125
126 block_type:
127   | NORMAL | START | END | CALL | RETURN | EXTERNAL | IMPASSE
```

```
128
129 edge:
130   | simple_item_lc_name_ctxt_edge_special
131
132 simple_item_lc_name_ctxt_edge_special:
133   | lc_id_def edge_special lc_name_ctxt_attr_list COLUMN
134   | lc_id_def edge_special COLUMN
135
136 edge_special:
137   | /*empty*/
138   | LPAREN edge_type RPAREN
139
140 edge_type:
141   | NORMAL | TRUE | FALSE | ZERO | DELAY | CALL | RETURN | IMPASSE
142
143 instruction:
144   | item_lc_name_ctxt_op_instr_special
145
146 item_lc_name_ctxt_op_instr_special:
147   | lc_id_def instr_special optional_lc_name_ctxt_attr_list COLUMN
148   | lc_id_def instr_special optional_lc_name_ctxt_attr_list LBRAC op_list RBR
149
150 op_list:
151   | op op_list
152   | op
153
154 instr_special:
155   | addr_and_width
156
157 addr_and_width:
158   | INT TWODOTS QUESTION_MARK
159   | QUESTION_MARK TWODOTS QUESTION_MARK
160   | QUESTION_MARK TWODOTS INT
161   | INT TWODOTS INT
162
163 op:
```

```
164   | simple_item_lc_name_ctxt_op_special
165
166 simple_item_lc_name_ctxt_op_special:
167   | lc_id_def op_special lc_name_ctxt_attr_list COLUMN
168   | lc_id_def op_special COLUMN
169
170 op_special:
171   | STRING
172
173 data:
174   | item_lc_name_bytes_data_special
175
176 item_lc_name_bytes_data_special:
177   | lc_id_def data_special optional_lc_name_attr_list COLUMN
178   | lc_id_def data_special optional_lc_name_attr_list LBRAC bytes_list RBRAC
179
180 optional_lc_name_attr_list:
181   | /*empty*/
182   | lc_name_attr_list
183
184 lc_name_attr_list:
185   | TWODOTS lc_name_comma_list
186
187 lc_name_comma_list:
188   | lc_name COMMA lc_name_comma_list
189   | lc_name
190
191 data_special:
192   |  dummy
193
194 bytes_list:
195   | bytes bytes_list
196   | bytes
197
198 bytes:
199   | simple_item_lc_name_bytes_special
```

```
200
201 bytes_special:
202   | /*empty*/
203   | addr_and_width
204
205
206 simple_item_lc_name_bytes_special:
207   | lc_id_def bytes_special COLUMN
208   | lc_id_def bytes_special lc_name_attr_list COLUMN
209
210
211 meta:
212   | item_lc_name_info_meta_special
213
214
215 item_lc_name_info_meta_special:
216   | lc_id_def meta_special COLUMN
217   | lc_id_def meta_special lc_name_attr_list COLUMN
218   | lc_id_def meta_special LBRAC info_list RBRAC
219   | lc_id_def meta_special lc_name_attr_list LBRAC info_list RBRAC
220
221 info_list:
222   | info info_list
223   | info
224
225 meta_special:
226   | dummy
227
228 info:
229   | simple_item_lc_name_info_special
230
231 simple_item_lc_name_info_special:
232   | lc_id_def info_special optional_lc_name_attr_list COLUMN
233
234 info_special:
235   | dummy
```

```
236
237 lc_name_ctxt:
238   | lc_name
239   | lc_name LESSTHAN context_ref GREATERTHAN
240
241 context_ref:
242   | lc_name
243   | STAR
244
245 context_def:
246   | CONTEXT lc_id_def TWODOTS optional_context_seq COLUMN
247
248 optional_context_seq:
249   | /*empty*/
250   | context_seq
251
252 context_seq:
253   | context_simple_comma_list
254
255 context_simple_comma_list:
256   | context_simple COMMA context_simple_comma_list
257   | context_simple
258
259 context_simple:
260   | context_match
261   | context_minimum
262   | context_repeat
263   | LPAREN context_seq RPAREN
264
265 context_match:
266   | lc_name RIGTH_ARROW lc_name
267   | QUESTION_MARK RIGTH_ARROW lc_name
268   | lc_name RIGTH_ARROW QUESTION_MARK
269   | QUESTION_MARK RIGTH_ARROW QUESTION_MARK
270
271 context_minimum:
```

```
272    | context_simple STAR
273    | context_simple STAR MINUS
274    | context_simple PLUS
275    | context_simple LBRAC INT COMMA RBRAC
276
277  context_repeat:
278    | context_simple QUESTION_MARK
279    | context_simple LBRAC INT COMMA INT RBRAC
280
281  value:
282    | value_numeric
283    | value_range
284    | value_date
285    | value_item
286    | value_vector
287    | value_map_symbol_value
288    | value_map_item_value
289    | value_functor
290
291  value_numeric:
292    | value_unsigned
293    | value_float
294
295  value_unsigned:
296    | INT
297
298  value_float:
299    | QUESTION_MARK FLOAT_KEYWORD RPAREN FLOAT LPAREN
300
301  value_range:
302    | DOT DOT
303    | value_numeric DOT DOT
304    | DOT DOT value_numeric
305    | value_numeric DOT DOT value_numeric
306
307  value_date:
```

```
308   | value_date_posix
309   | value_date_humal
310
311 value_date_posix:
312   | QUESTION_MARK GMT LPAREN INT RPAREN
313
314 value_date_humal:
315   | QUESTION_MARK GMT LPAREN INT MINUS INT MINUS INT MINUS INT MINUS INT MINUS IN
316
317 value_item:
318   | lc_name
319
320 value_vector:
321   | LLPAREN RRPAREN
322   | LLPAREN vector_list RRPAREN
323
324 vector_list:
325   | vector_element_comma_list
326
327 vector_element_comma_list:
328   | vector_element COMMA vector_element_comma_list
329   | vector_element
330
331 vector_element:
332   | value
333   | INT EQUALS value
334
335 value_map_symbol_value:
336   | LBRAC RBRAC
337   | LBRAC symbol_map RBRAC
338
339 symbol_map:
340   | symbol_map_entry_comma_list
341
342 symbol_map_entry_comma_list:
343   | symbol_map_entry COMMA symbol_map_entry_comma_list
```

```
344    | symbol_map_entry
345
346  symbol_map_entry:
347    | symbol_map_key
348    | symbol_map_key EQUALS value
349
350  symbol_map_key:
351    | lc_name
352    | STRING
353    | INT
354
355  value_map_item_value:
356    | AT LBRAC RBRAC
357    | AT LBRAC item_map RBRAC
358
359  item_map:
360    | item_map_entry_comma_list
361
362  item_map_entry_comma_list:
363    | item_map_entry COMMA item_map_entry_comma_list
364    | item_map_entry
365
366  item_map_entry:
367    | item_map_key
368    | item_map_key EQUALS value
369
370  item_map_key:
371    | lc_name
372
373  value_functor:
374    | functor_c
375    | functor_mathematica
376    | functor_lisp
377    | functor_prefix
378    | functor_infix
379    | functor_circumfix
```

```
380    | functor_suffix
381
382  functor_c:
383    | identifier LPAREN RPAREN
384    | identifier LPAREN vector_list RPAREN
385
386  functor_mathematica:
387    | identifier LLPAREN RRPAREN
388    | identifier LLPAREN vector_list RRPAREN
389
390  functor_lisp:
391    | LPAREN identifier RPAREN
392    | LPAREN identifier COMMA vector_list RPAREN
393
394  functor_prefix:
395    | LPAREN identifier value_vector RPAREN
396
397  functor_infix:
398    | LPAREN value_vector identifier value_vector RPAREN
399
400  functor_circumfix:
401    | LPAREN identifier value_vector identifier RPAREN
402
403  functor_suffix:
404    | LPAREN value_vector identifier RPAREN
405
406  identifier:
407    | IDENT
```