Universität des Saarlandes
Fachbereich 6.2 Informatik
Lehrstuhl für Programmiersprachen und Übersetzerbau


# Static Timing Analysis Tool Validation in the Presence of Timing Anomalies

**Dissertation**

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

von
Gernot Gebhard

Saarbrücken
2013

| | |
|---|---|
| Dekan | Prof. Dr. Mark Groves |

## Prüfungsausschuss

| | |
|---|---|
| Vorsitzender | Prof. Dr. Jan Reineke |
| Berichterstatter | Prof. Dr. Dr. h.c. mult. Reinhard Wilhelm |
| | Prof. Dr.-Ing. Wolfgang Kunz |
| Akademischer Beisitzer | Dr.-Ing. Michael Feld |

| | |
|---|---|
| Tag des Kolloquiums | 22.10.2013 |

# Eidesstattliche Versicherung

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

Saarbrücken, 7. November 2013

# Zusammenfassung

Um das Zeitverhalten eines sicherheitskritischen eingebettenen Softwaresystems zu validieren, benötigt man sichere und präzise Grenzen für die Ausführungszeiten der einzelnen Softwaretasks im schlimmsten Falle (Worst-Case). Diese Zeitschranken müssen zuverlässig sein, damit sichergestellt ist, dass jede Komponente des Softwaresystems rechtzeitig ausgeführt wird. Zudem müssen die zuvor bestimmten Zeitschranken so präsize wie möglich sein damit das Softwaresystem als Ganzes (beweisbar) ausführbar ist (Schedulability). Für die Erreichung dieser beiden Ziele stellen Zeitanomalien eine der größten Hürden dar. Fast jede moderne Prozessorarchitektur weist Zeitanomalien auf, die einen großen Einfluß auf die Analysierbarkeit solcher Architekturen haben.

Eine Zeitanomalie ist ein kontraintuitives Verhalten einer Hardwarearchitektur, bei dem ein lokal *gutes* Ereignis (z.B., ein Cache Hit) zu einer insgesamt längeren Ausführungszeit führt, das entgegengesetzte *schlechte* Ereignis (in diesem Fall ein Cache Miss) aber eine global kürzere Ausführungszeit mit sich bringt. Weist eine Prozessorarchitektur ein solches Verhalten auf, darf eine Zeitanalyse für diese Architektur nicht nur lokal schlechte Ereignisse in Betracht ziehen, um eine obere Schranke der worst-case Ausführungszeit für einen Task zu ermitteln. Um zuverlässige Zeitgarantien zu bestimmen, muss eine Zeitanalyse alle möglichen Ausführungszustände betrachten, die durch unbekannte Hardwarezustände entstehen könnten.

In dieser Arbeit untersuchen wir die Ursache von Zeitanomalien in modernen Prozessorarchitekturen und betrachten Zeitanomalien, die auch in eher einfachen Prozessoren vorkommen können. Desweiteren diskutieren wir den Einfluß von Zeitanomalien auf statische Zeitanalysen für eben solche Architekturen, die dieses nicht-lokale Zeitverhalten aufweisen. Zuletzt zeigen wir, wie mittels Trace Validierung Analyseergebnisse von statischen Zeitanalysen in diesem Kontext überprüft werden können.

# Abstract

The validation of the timing behavior of a safety-critical embedded software system requires both safe and precise worst-case execution time bounds for the tasks of that system. Such bounds need to be safe to ensure that each component of the software system performs its job in time. Furthermore, the execution time bounds are required to be precise to ensure the (provable) schedulability of the software system. When trying to achieve both safe and precise bounds, timing anomalies are one of the greatest challenges to overcome. Almost every modern hardware architecture shows timing anomalies, which also greatly impacts the analyzability of such architectures with respect to timing.

Intuitively spoken, a timing anomaly is a counterintuitive behavior of a hardware architecture, where a *good* event (e.g., a cache hit) leads to an overall longer execution, whereas the corresponding *bad* event (in this case, a cache miss) leads to a globally shorter execution time. In the presence of such anomalies, the local worst-case is not always a safe assumption in static timing analysis. To compute safe timing guarantees, any (static) timing analysis has to consider all possible executions.

In this thesis we investigate the source of timing anomalies in modern architectures and study instances of timing anomalies found in rather simple hardware architectures. Furthermore we discuss the impact of timing anomalies on static timing analysis. Finally we provide means to validate the result of static timing analysis for such architectures through trace validation.

# Acknowledgments

First of all, I wish to thank Prof. Dr. Reinhard Wilhelm for giving me the opportunity to conduct this work. We have had important and fruitful discussions that helped me in improving this thesis a lot. Additionally, I thank Prof. Dr. Reinhard Wilhelm and Prof. Dr. Wolfgang Kunz in advance for the survey of this work.

I am thankful for the time granted at AbsInt to implement and to evaluate the timing anomaly instance detector and the trace validation framework during my everyday work schedule. Special thanks go to Christian Ferdinand, Christoph Cullmann, Florian Martin, Reinhold Heckmann, Markus Pister, Marc Schlickling, Jan Reineke, and Heiko Falk for many helpful comments about various aspects of this thesis. They all have had great influence on the style of this document.

Last but not least, I thank my friends, my whole family and my wife Tina for their support, without which I would not have been able to complete this work after all.

Extra credits go to all authors of free software for their excellent work. This thesis has been written entirely using free software.

# About this Document

I have written this document using Kate [17], LaTeX [19], and Rubber [36]. The PDF features clickable cross-references that connect references to their origin. In this fashion, the reader can easily jump to a footnote, a figure, or a citation, among others, wherever referenced. Although both figures and tables are (intended to be) self-explaining most of the time, I advice the reader to always parse them in conjunction with the surrounding text to prevent any misconceptions.

# Contents

# 1 Introduction

Embedded systems are omni-present. For example, modern cars comprise a multitude of embedded systems, some of which perform safety-critical operations, as e.g., adaptive cruise control, anti-lock braking system, or airbag control. Besides functional correctness, failure to react within a certain time budget might result in fatal consequences. Hence, the verification of the system's timing behavior plays a key role during its development. To validate the timing behavior and to guarantee schedulability of the overall system, both safe and precise execution time bounds are required for each system component (i.e., task). Knowing the worst-case execution time (WCET) is of major interest.

Determining precise timing bounds is however not easily achieved because a task's execution time greatly depends on its inputs, its mode of operation, the initial hardware state, and the interference with the environment. For instance, different operating modes of a flight control unit, such as *plane is on ground* and *plane is in air*, could lead to mutual exclusive execution paths that exhibit different timing behavior. With a high probability, measurements will observe average-case performance and might only by chance encounter the worst-case timing behavior or the best-case timing behavior respectively of the program (see Figure 1.1).

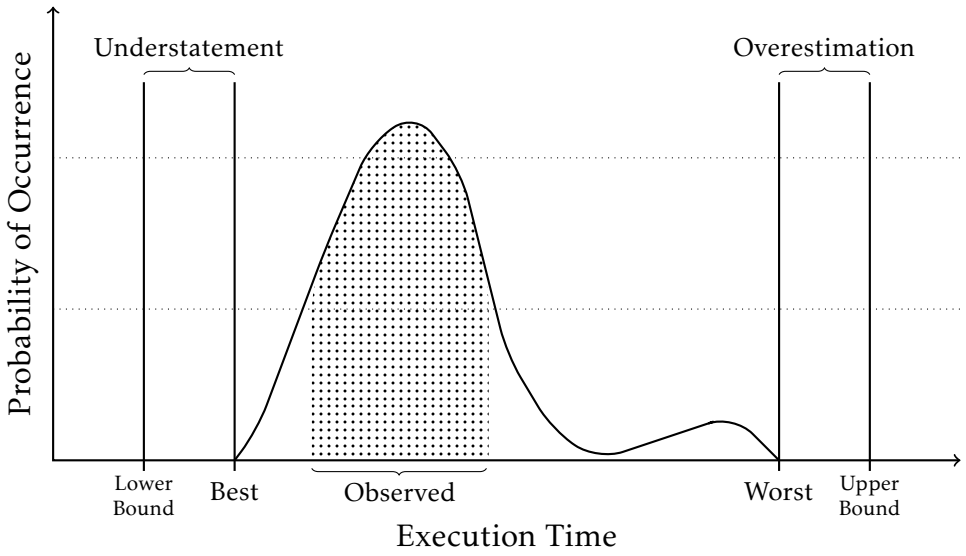



Figure 1.1: *Execution Time Distribution of a Task:* Measurements will most likely deliver average-case execution times. The best-case or the worst-case timing behavior is seldom observed.

It becomes obvious that measurement-based timing analysis approaches cannot succeed in determining the WCET of a piece of software. This is also due to the fact that computing an initial system state that triggers the worst-case timing behavior is not easily achieved.

Static timing analyses attempt to estimate worst-case timing behavior without actually executing the software on the real hardware. In general, the state space of input data and initial states is too large to exhaustively explore all possible executions in order to determine the exact worst-case and best-case execution times. Instead, static timing analysis can only provide lower and upper bounds for the whole system's execution time, as shown in Figure 1.1.

Some abstraction of the execution platform is necessary to make a timing analysis of the system feasible. These abstractions lose information, and thus are – in part – responsible for the gap between WCET guarantees and observed upper bounds. How much is lost depends on the methods used for timing analysis and on system properties, such as the hardware architecture and the analyzability of the software. Even despite the potential loss of precision caused by abstraction, static timing analysis methods are well established in the industrial process [42].

In order to achieve the required performance, processors exhibit features to increase average-case-performance. Some of these features might provoke timing anomalies. A timing anomaly is a counterintuitive behavior of a hardware architecture, where a *good* event (e.g., a cache hit) leads to an overall longer execution, whereas the corresponding *bad* event (e.g., a cache miss) leads to a globally shorter execution time. In the presence of such anomalies, the local worst-case (i.e., the bad event) is not always a safe assumption in static timing analysis. To compute safe timing guarantees, any static timing analysis then has to consider all possible executions. Due to the loss of predictability, a static analysis of such architectures requires much more effort in terms of computing power and memory consumption. Computing tight timing bounds is a challenge.

In this thesis we investigate the source of timing anomalies in modern architectures and study timing anomalies found in rather simple hardware architectures. Furthermore we discuss the impact of timing anomalies on static timing analysis for architectures that exhibit this kind of non-local timing behavior. Finally we provide means to validate the result of static timing analysis through trace validation.

## 1.1 Embedded Hardware Architectures

Ever since the invention of the microprocessor at Intel in 1971 the computing power has been steadily increasing in accordance with Moore's Law [27]. However, as stated by Mahapatra et al. [25] the rate of improvement in processor speed exceeds the rate

of improvement in memory speed. Due to technological limitations memory speed is roughly increasing at a rate of 10% per year. This rate of development is rather slower compared to an average speed increase of central processing units (CPUs) of about 60% per year. Hence, system designers face an increasing processor-memory performance gap. Figure 1.2 depicts this issue in more detail.

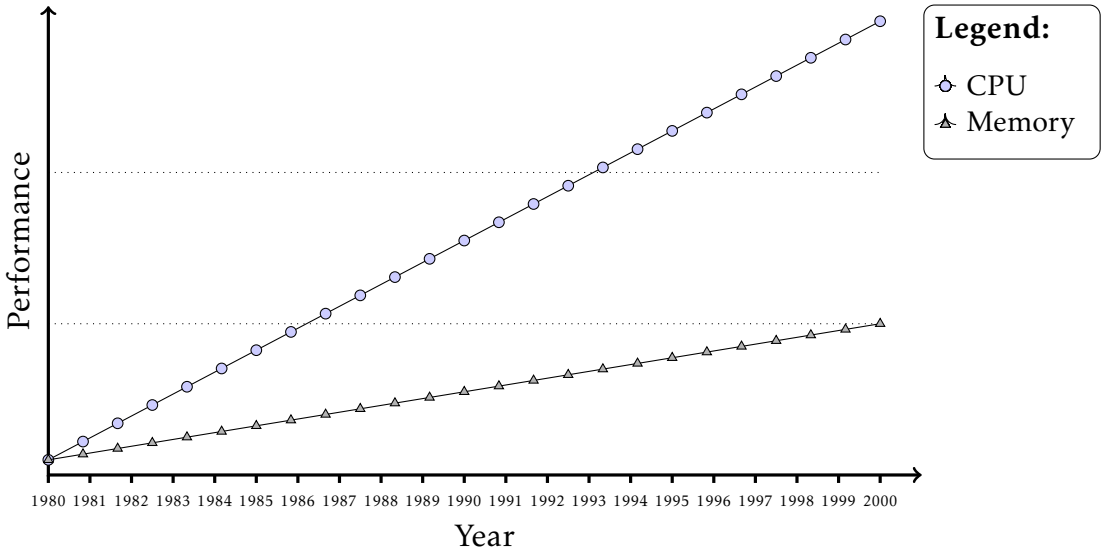



Figure 1.2: *Processor-Memory Performance Gap:* Starting with the performance found in 1980 the gap between CPU and memory processing speed has been ever increasing [25].

The memory architecture of an (embedded) processor is structured hierarchically from fast but small memory to slow but large memory, as shown in Figure 1.3. The memory hierarchy concept benefits from the principle of locality, which states that instruction and data that are located close to each other tend to be referenced close together in time (*spatial locality*), and that recently accessed memory blocks are likely to be accessed again (*temporal locality*) [12].

## 1.1.1 Memory Hierarchy

This section discusses types of memory typically found in embedded hardware and investigates their influence on the timing behavior.

### Register File

The register file is a small set of memory cells in which the CPU stores the results of executed operations. The compiler decides which variables are (temporarily) stored in the registers. The register allocation significantly impacts the performance of an

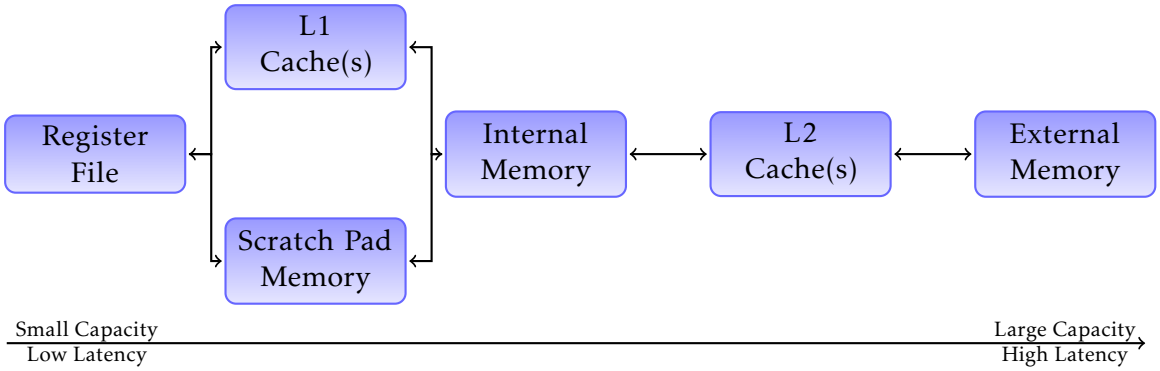



Figure 1.3: *Memory Hierarchy of an Embedded Processor:* Different levels of memory hierarchy in an embedded hardware architecture. Both capacity and access latency increase with the distance to the register file.

application. Less spilling code will lead to a smaller code size and to less memory accesses that load and store register contents from and to main memory.

### Cache Memory

Caches are small, fast memories that transparently load and store memory blocks from and to main memory. If the cache contains a memory block that is currently requested by the core (*cache hit*) it can be served quickly. Otherwise if the cache does not contain the requested memory block (*cache miss*), it then decides in accordance with a replacement policy which of the previously cached memory blocks to evict from the cache. Often found replacement policies in embedded architectures are first-in-first-out (FIFO), least-recently-used (LRU), pseudo-round-robin, or pseudo-LRU (an approximation of LRU that requires less bits in hardware implementation). The replacement policy has a great influence on the overall predictability of the entire system. Obviously, caches with a random replacement policy are far less predictable than those using the LRU replacement strategy. Unfortunately, hardware architects prefer less predictable replacement policies, as these are less costly to implement.[1]

Caches vary in their configuration, which is usually measured in cache line size (i.e., size of the memory blocks to store), number of cache sets, the associativity (i.e., the number of blocks to associate with one cache set), and the overall cache size. Caches are further discerned into *unified caches* that store both code and data and *disjoint caches* that store code and data separately. For static timing analysis the use of unified caches usually leads to a decrease in predictability.

[1] The LRU replacement policy requires a rather complex update logic, which results in higher hardware cost, power consumption, and thermal output.

Modern embedded architectures already feature two levels of caches to bridge the performance gap between internal and external memories. The use of several levels of caches however furthermore impairs the architecture's predictability.

### Scratch Pad Memory

Scratch pad memories are fast and small memories that are tightly coupled to the CPU core. But unlike caches, scratch pad memories are software-managed. This means that the programmer or the compiler has to decide which memory blocks should be put into the scratch pad. The manual decision of which data to keep in the scratch pad memory can be beneficial in the overall context, despite the overhead caused by the scratch pad memory update.

The TriCore TC1797 processor features a 40KB of code scratch pad memory of which 16KB are shared with the instruction cache [40]. The system developer may decide upon the scratch pad and cache partitioning (see Table 1.1).

### Internal Memory

The internal memory is often subdivided into code storage and data (e.g., amongst others, stack) storage. The code is stored in electrically erasable programmable read-only memory (EEPROM) or Flash memory. Most architectures provide a small code prefetch buffer (usually up to four cache lines) to further speed up accesses to the code storage. Program data is located in a separate static random-access memory (SRAM) region.

In a Freescale MPC5554 processor the internal Flash memory stores up to 2MB of code, the internal SRAM memory at most 64KB of data. For comparison, the MPC5554 has a unified cache storing 32KB of code and data [28].

| Scratch Pad Size (KB) | Instruction Cache Size (KB) |
|---|---|
| 24 | 16 |
| 32 | 8 |
| 36 | 4 |
| 38 | 2 |
| 40 | 0 |

Table 1.1: *Partitioning of Scratch Pad and Instruction Cache in a TC1797 CPU:* The user may choose between several configurations. By default the full scratch pad size is configured.

#### External Memory

External memory provides additional storage capacity for data (or even code if internal memory space is too small). Here dynamic random-access memory (DRAM) or synchronous dynamic random-access memory (SDRAM) provides larger capacities (compared to SRAM) at an acceptable cost. The dynamic design of DRAM/SDRAM and the larger access time implies the reduced performance (and predictability) as compared to EEPROM or SRAM.

### 1.1.2 Processor Features

The memories and the memory hierarchy present in an embedded hardware architecture have a huge influence on the system's (worst-case) timing behavior. Furthermore, other performance-enhancing features are implemented to improve the processor's execution time behavior. As we will show throughout this thesis, the combination of performance-enhancing features often leads to timing anomalies, which impair the predictability of the overall system.

#### Pipelining

The execution of an instruction is subdivided into several execution phases, called *pipeline stages*. In principle pipeline stages may operate independently from each other as long as there are no intra-instruction dependencies. For example, the processor may decode the current instruction, while executing the previously fetched one. Upon a dependency between two instructions, the processor inserts stall cycles (*bubbles*) until this dependency is resolved. We discern the following pipeline stages:

**Fetch** The instruction is fetched from main memory or from the instruction cache and is placed into a dedicated instruction buffer.

**Decode** The operation code is decoded. In this phase, the processor determines which registers are read and written by the instruction to execute.

**Execute** The instruction is brought to execution. Required register contents are read from the register file. Depending on the kind of instruction the processor takes some time to compute the result. For instructions that access the memory the processor computes the target address (*effective address calculation*).

**Memory** Instructions that access main memory issue their request to the memory controller.

**Write back** The instruction result is finally written into the register file.

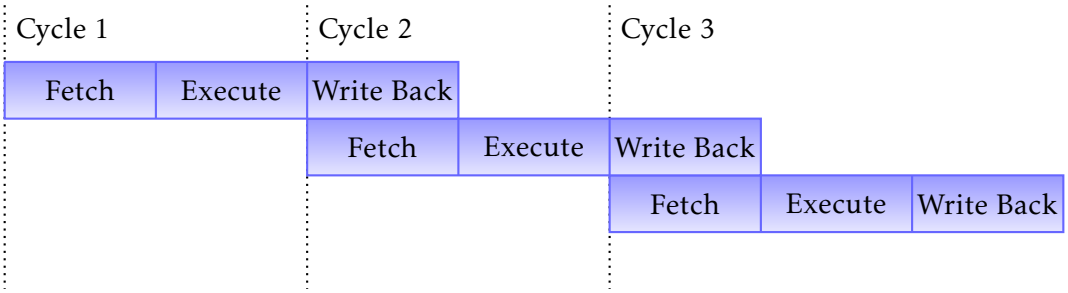



Figure 1.4: *Zuse Z3 Pipeline:* Simple in-order pipeline. The instruction fetch stage overlaps with the write back stage.

Even the Zuse Z3 featured a simple in-order pipeline [35]. Figure 1.4 shows the pipeline of the Z3, where the write back stage overlaps with the fetch stage.

We discern between structural dependencies, i.e., two instructions requesting the same functional unit, and data dependencies. Data dependencies are further classified into the following classes:

**read after write** An instruction reads a register that is being written by a preceding instruction. The processor then has to stall execution until the result of the preceding instruction is available. This is the only true dependency between two instructions.

**write after read** An instruction attempts to write to a register before it is being read by another instruction. This dependency can only occur in out-of-order execution pipelines, where two instructions may execute in parallel.

**write after write** Two instructions attempt to write to the same register. To guarantee correct program semantics, the processor has to ensure that the register content is updated in the correct order. This dependency does not exist for in-order completion pipelines.

Read after write dependencies may cause *control dependencies*. Figure 1.5 shows a simple example. The processor is waiting for the outcome of the branch and stalls the fetch until the branch target is known. There are several approaches not to stall the code fetch as discussed below.

The length of the processor pipeline can become an issue for static timing analysis. The amount of data to maintain during static analysis grows linearly in the number of pipeline stages, which is problematic in terms of memory consumption.

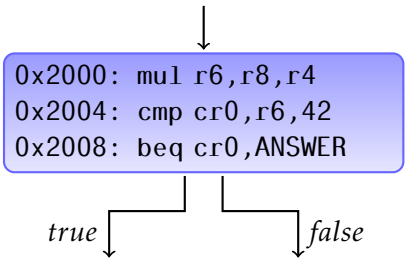



Figure 1.5: *Control Dependency:* We assume an in-order five-stage pipeline. The `mul` instruction resides in the execute stage and waits for completion. The `cmp` compare instruction depends on the outcome of the multiplication and is stalled in the decode stage. Finally, the fetch stage contains a branch instruction depending on the comparison. At this point, the processor does not know whether the *true* of *false* branch target should be fetched next.

### Forwarding

A processor can partially resolve read after write dependencies by *forwarding* intermediate results. With this technique the processor can start the execution of an instruction even if the register result has not yet been written back. This allows a processor to dispatch instructions completing in one cycle one after another, even if they suffer from register dependencies.

### Early-Out Execution

Early-out execution allows for a better average-case performance. The number of cycles to compute a result may depend on the instruction operands. For instance, the execution of a division operation can be accelerated if the denominator is 0, or if the denominator is 1.

Varying instruction execution times pose a problem to static analysis. In the presence of timing anomalies, it is not safe to only assume the worst execution time possible. Depending on the input program taking into account all possibilities could render a precise static timing analysis infeasible due to exponential memory consumption.

### Prefetching

Prefetching is a mechanism to avoid stalling the fetch stage. Instead of waiting for a branch outcome, the processor prefetches consecutive instructions and puts them into a dedicated prefetch buffer. Prefetched instructions are not executed until the branch outcome is available. This is not to be confused with speculative execution, where prefetched instructions are brought to execution whereas their results are not written to the register file (see below). Depending on the branch condition, the

processor either continues executing instructions from the prefetch buffer or clears the prefetch buffer and starts fetching from the branch target.

The prefetch buffer size is constantly increasing. The Freescale MPC555 processor comprises a two-instruction prefetch buffer. The recent MPC7448 stores up to 12 instructions in the prefetch buffer.

A static analysis for the prefetching mechanism requires minimal effort, because the processor behavior is always deterministic and the amount of additional data to remember is rather limited.[2] In the presence of an instruction cache, the prefetching mechanism can lead to cache pollution and even a lower performance compared to not prefetching (depending on the cache line fill costs). Chapter 4 shows that this combination can lead to a timing anomaly.

### Branch Prediction

Branch prediction is an extension of the simple prefetching mechanism discussed above. In order to reduce the number of useless instruction fetches, the processor employs a heuristic to determine whether the branch target is to be fetched next. Embedded processors either use *static branch prediction* or *dynamic branch prediction*.

Static branch prediction discerns between *forward branches* and *backwards branches*. A branch is a forward branch if the branch target is located in memory after the branch instruction, i.e., $address(branch\ instruction) < address(branch\ target)$. Otherwise the branch is called backwards branch. Forward branches are statically predicted not taken. Encountering a forward branch, the processor continues to fetch linearly. The compiler generates forward branches to implement if-then-else constructs. After fetching a backwards branch, the processor prefetches in the direction of the branch target. Backwards branches are usually generated for loops, where it is more likely that the control flow continues with the branch target. If supported by the processor's instruction set architecture, the compiler may influence the direction of prediction (*prediction bit*).

Dynamic branch prediction estimates the direction of control flow by means of the branch execution history, which is stored in the branch history table (BHT). For a branch instruction, the BHT stores a $k$-bit saturating counter that indicates how likely the branch is being taken. For instance, the Freescale MPC7448 features a 2048-entry BHT using 2 bits per entry for four levels of prediction: *strongly not taken*, *weakly not taken*, *weakly taken*, and *strongly taken*. Upon a taken branch, the corresponding counter is incremented, i.e., it is more likely to be taken, and decremented vice versa. In addition to the BHT, such processors are often equipped

[2] For this prefetch mechanism, it suffices to remember the number of additionally prefetched instructions, which is bounded by the size of the prefetch buffer.

with a branch target buffer (BTB), or branch target instruction cache (BTIC), that stores the target instructions of recently taken branches. If a target instruction is present in the BTIC it is directly made available to the prefetch buffer. This can happen faster than accessing the instruction from the instruction cache. The MPC7448 comprises a 32-set, four-way set associative BTIC that is updated using a FIFO replacement strategy.

Static branch prediction does not pose much of a problem to static timing analysis. Like simple prefetching, the processor's behavior can always be determined precisely. Dynamic branch prediction is far more complex to analyze statically. The key problem is on the one hand the large size of the branch history table, and on the other hand the usage of statically less predictable replacement polices that are often prone to timing anomalies, such as FIFO. For complex processors, a precise analysis of the BHT and the BTIC is practically infeasible due to the overly large state space. Hence, disabling dynamic branch prediction allows static timing analyses to provide far more precise WCET bounds. However, for the MPC5xx processor derivatives, we have successfully implemented an analysis for their BTIC [15].[3]

### Delay Slots

Delay slots are used to implement delayed branches. Before executing a branch instruction the processor first executes the instructions in the associated delay slots. This is done independently from the branch outcome, allowing the processor to perform useful operations while executing the branch. The delay slot size differs between hardware architectures. PowerPC and X86 architectures do not feature delay slots. SPARC architectures, such as LEON2, or LEON3, have a single delay slot. The SHARC digital signal processor has two delay slots. The C33 processor features three delay slots.

Delay slots can raise problems during the decoding of the program binary. Figure 1.6 shows an example, where the compiler has put the load of a volatile variable inside a delay slot. Due to this optimization, the decoder has to discern two separate loops even though the executed instructions are semantically equivalent, independent of which path through the loop is being taken.

### Superscalar Architecture

A *superscalar processor* dispatches multiple instructions at once onto several functional units (e.g., arithmetic logical units). In this fashion the processor is able to execute more than one instruction per cycle. The number of dispatched instructions

[3] The MPC5xx processors do not possess a branch history table.

```
extern volatile bool busy;
int worker (Job *job) {
  int result = 0;
  while (busy) {
    Task *task = job->task;
    if (task != NULL)
      result = performTask (task);
  }
  return result;
}
```

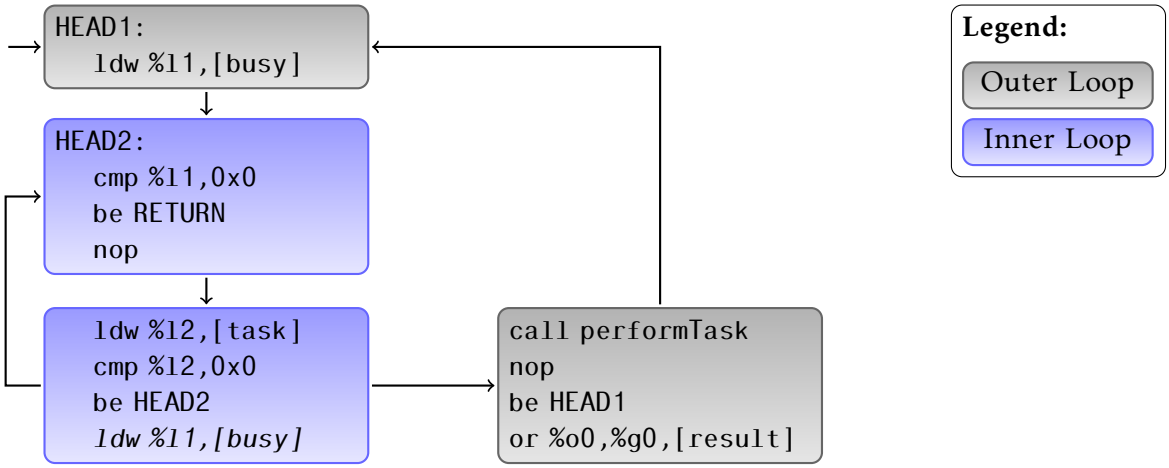



Figure 1.6: *SPARC Delay Slot Optimization:* The load of the volatile variable busy is also performed in the delay slot (in italics) of the second branch instruction (i.e., `task != NULL`). By doing so, the compiler has transformed a single loop into two nested loops. This optimization has been found in a satellite control software system.

per processor cycle depends on the stream of instructions and their interdependencies. This behavior complicates the abstract processor model due to the dynamic nature of the dispatch mechanism.

For instance, the Freescale MPC755 processor is able dispatch two instructions per cycle. However, the maximum performance of two instructions per cycle is rarely achieved due to dependencies between adjacent instructions.

### Out-of-Order Execution

To limit the effect of data and structural dependencies instruction dispatch may be executed out-of-order. This mechanism allows for a higher utilization of the processor's functional units because instructions with no dependencies can be

executed earlier than in a strict in-order pipeline design. The functional units may be used out of order as well. For instance, a floating point operation usually takes longer than an integer operation. An integer addition might complete earlier than a preceding floating point division.

Similarly to a superscalar architecture design, achieving a precise static timing analysis is complicated due to the dynamic behavior of out-of-order execution.

#### Speculative Execution

Speculative execution attempts to further reduce the cost of conditional change-of-flow instructions by executing prefetched instructions ahead of time. The results are stored in intermediate registers and have to be discarded in case the branch prediction speculated in the wrong direction. As long as the corresponding branch outcome is unknown, the speculative results may not be written back to the register file.

For example, the MPC7448 supports up to three outstanding speculative branches. Speculative execution is also implemented in 16bit architectures, such as the Infineon C166V2.

For static timing analysis, speculative execution can impair the analysis precision in case the analysis cannot determine precisely whether speculation is applied. This could occur directly after the execution of an instruction with varying execution time, such as an integer multiplication (see Figure 1.5). At worst, the analysis can gather only little information about potential instruction or data cache contents.

### 1.1.3 Timing Anomalies

Lundqvist and Stenström [24] have discovered the effect of *timing anomalies*. They state that it does not suffice to only investigate local worst-case behavior of individual instructions to determine a global worst-case execution time bound. In their paper the authors provide an example of a timing anomaly, where a cache hit leads to the worst-case timing.

Engblom and Jonsson [9] also discuss timing anomalies. They translate the notion of timing anomaly of Lundqvist and Stenström [24] to their model considering (local) timing of pipeline stages instead of whole instructions. Both Lundqvist and Engblom claim that timing anomalies cannot occur in processors that only comprise in-order resources (i.e., two instructions can only use a resource in program order).

Wenzel et al. [45] extend the work of Lundqvist and Engblom providing a necessary, but not sufficient, condition for the occurrence of timing anomalies in superscalar processors. Demonstrating a timing anomaly by means of an artificial hardware model with in-order resources only, they are able to refute Lundqvist's and Engblom's

claim. However, the presented necessary condition is limited to architectures that have at least two functional units with different timing behavior. In this thesis we show that even in-order architectures with a single functional unit may suffer from timing anomalies.

Schneider [38] reveals that the instruction scheduling mechanism in the Freescale MPC755 processor is prone to a timing anomaly. The possibility to dispatch an instruction on two execution units with different timing behavior in conjunction with pipeline stalls can trigger a timing anomaly. Furthermore, he was the first to provide a real example for a *domino effect* [24], which is some kind of non-stabilizing timing anomaly. Thesing [42] discusses the Motorola ColdFire 5307 that has a rather simple in-order pipeline. He shows that the processor exhibits domino effects, caused by the pseudo round-robin cache replacement algorithm.

Berg [3] discusses cache replacement policies and their timing anomalies. He finds that caches using first-in first-out (FIFO), round-robin, or pseudo least-recently-used (PLRU) cache replacement strategies suffer from timing anomalies. These replacement strategies are commonly used in embedded hardware architectures, as they require less update logic compared to the LRU policy, which is free of timing anomalies.

At the time of writing there exists no method that automatically detects timing anomalies. Every timing anomaly that has been found so far was determined manually. Eisinger et al. [8] have provided a novel methodology to automatically detect timing anomalies. Requiring an accurate hardware model to be available (e.g., in VHDL), the approach attempts to synthesize an instruction sequence that triggers a timing anomaly if such a sequence exists. Yet, the approach is not fully automatic because hardware features potentially causing timing anomalies need to be identified manually.

Reineke and Sen [33] discuss a related method that allows a static timing analysis to safely discard analysis states by means of $\Delta$ functions. A $\Delta$ function computes the maximal difference in timing between two system states on any input instruction sequence. For any pair of system states, a static timing analysis can consult the corresponding $\Delta$ function to determine which of the two states can be safely discarded. This approach works quite well for simple hardware models with a very limited instruction set architecture (ISA). However, there exists no feasible algorithm to compute $\Delta$ functions for real hardware architectures.

Kirner et al. [18] show that splitting up a WCET analysis into separate parallel WCET analyses (corresponding to hardware components operating in parallel) is not generally safe in the presence of timing anomalies. Furthermore, the authors identify special instances of *parallel* timing anomalies still making a parallel decomposition of the WCET problem feasible. These findings correspond to the classification of

architectures discussed in Chapter 4. Non-fully timing compositional architectures do not allow for a safe, parallel decomposition of the WCET problem.

Reineke et al. [34] are the first to provide a formal definition of timing anomalies in the context of worst-case execution time analysis. Chapter 6 on page 73 adopts this definition and extends the formalism to describe domino effects.

## 1.2 Thesis Structure

Chapter 2 provides the basic formalism used throughout this thesis with a strong focus on abstract interpretation. Chapter 3 discusses timing analysis in general and briefly depicts the structure of state-of-the-art static timing analyzers. The remainder of the chapter is concerned with the description of static pipeline analyses. We discuss the types of imprecision that arise during pipeline analysis and correlate them with hardware features, where possible. Finally, we formally define the functioning of static pipeline analysis. Chapter 4 adopts the formalism introduced in Chapter 3 to precisely define the terms timing anomaly and domino effects. Furthermore, we discuss recently found instances and investigate their causes. We also show how instances of timing anomalies can be semi-automatically detected and visualized by means of static timing analysis results. Chapter 5 demonstrates how static timing analysis results can be combined with measurements. We provide means to verify the safety of static timing analysis. Chapter 6 evaluates the findings made. Finally, Chapter 7 concludes the thesis.

# Program Analysis 2

Is the execution of a specific function in a program possible? Will the fuel rods in a nuclear power plant retract fast enough to prevent a meltdown? Answering such questions is one of the main purposes of formal verification through program analysis. Especially in the domain of safety-critical applications, developing correct programs is of major interest. Failure to do so might result in damage to equipment, harm or, even worse, loss of life, or long term environmental damage. For instance, (static) program analysis could have prevented the Ariane 5 maiden flight crash in 1996, which was caused by an arithmetic overflow during the conversion of a 64bit floating point number into a 16bit integer value [21].

Dynamic program analyses attempt to determine program properties through (not necessarily exhaustive) testing. The program is executed under a multitude of input conditions. Such analysis methods cannot provide absolute guarantees as we will later show.

Static analyses compute program characteristics without executing the program. Either the program source code, or the corresponding object code, or some other representation serves as input to the analysis. Several static analysis techniques have emerged so far. *Model checking* attempts to derive by means of a system model and given specification whether the model meets the specification. With increasing complexity of such models, the model checking approach cannot be applied to real world applications in a beneficial way. *Abstract interpretation* uses an abstract system model to simulate the system behavior on a set of abstract states. This approach allows (static) program analysis even of sophisticated systems. A static analysis based on abstract interpretation computes an over-approximation of the set of possible concrete values. Contrary to dynamic program analysis, analyses based on abstract interpretation can be designed to provide safe guarantees.

For the remainder of this document, we assume that the reader is familiar with lattice theory [7] and with the principles of program analysis [30]. In Section 2.1 we introduce abstract interpretation of programs with a focus only on the formalisms required by this thesis.

## 2.1 Abstract Interpretation of Programs

Abstract interpretation was formalized by Patrick and Radhia Cousot [5, 6]. The basic concept of abstract interpretation is to discard irrelevant information about the concrete system semantics and to keep what is relevant to verify the specification.

| Statement | Source Code | Interval Abstraction for y | Sign Abstraction for y |
|---|---|---|---|
| 1 | `if (x > 0)` | $\top$ | $\top$ |
| 2 | `  y = 4;` | 4 | $\{pos\}$ |
| 3 | `else` | $\top$ | $\top$ |
| 4 | `  y = -4;` | -4 | $\{neg\}$ |
| 5 | `x = x/y;` | ***[-4, ..., 4]*** | $\{pos, neg\}$ |

Table 2.1: *Division by Zero Runtime Error Detection:* Statements which (might) cause a division by zero error are in italics. Using the interval domain to abstract from concrete values of y, the analysis is unable to prove the absence of a runtime error. On the contrary, using the sign abstraction, the analysis can prove that no error occurs.

For example, to compute a (worst-case) execution time bound for an instruction sequence it is not necessary to keep track of the register contents. An addition instruction will (typically) compute its result after a fixed number of processor cycles, independent of the input values.[4]

The chosen abstraction should be *sound*. This means a proof stating that the abstract semantics satisfies the abstract specification should imply that the concrete semantics also satisfies the concrete specification. Hence, testing is not a sound verification method because it cannot be guaranteed that all possible executions are included. Unsound abstractions lead to *false negatives*, i.e., a program is claimed to be correct, although it actually does not behave according to its specification.

Furthermore, it is desirable to chose a *complete* abstraction. This means that if the concrete program semantics satisfies its specification this has to be confirmable by the abstract semantics. However, as program proofs are undecidable in general, any kind of (static) analyzer for non-trivial program properties is inevitably incomplete. Hence, a static analysis may emit *false positives* or *false alarms*, i.e., a violation against the specification is detected even though it cannot occur in reality.

Table 2.1 gives an example for an incomplete abstraction. Using an interval domain to abstract from the real values the variable y can accept, the analysis is unable to prove that no division by zero runtime error can occur at the last program statement.

Finding good abstractions is a key task in the design of analyses based on abstract interpretation. Consider again the example in Table 2.1. By using the sign domain (i.e., a value can have any combination of the properties *positive*, *negative*, or *zero*) the

[4] For instructions with early-out execution behavior this is not quite true. The analysis can achieve a better precision by modeling the early-out mechanism.

analysis is able to prove that a division by zero cannot occur. For the last statement, the variable y is known to be not zero. However, a sign domain based analysis does not always outperform an interval domain based analysis. If the divisor would be computed by an addition of a positive and a negative value, the sign domain based analysis could not prove the absence of an error. Depending on the abstract values of the terms of the sum, the interval domain based analysis would then still be able to show that a division by zero cannot happen.

Abstractions cannot be arbitrarily precise for computability reasons. Hence, there is always a trade-off between achievable analysis precision and required computational resources. For certain systems it is even impossible to design an abstraction that performs significantly better than a naive approach. For example, there cannot be a better abstraction for a cache with a random replacement policy than the one that only remembers the last accessed set. However, this issue is out of scope for this thesis.

Section 2.1.1 formally defines the used program representation and introduces the concrete execution behavior. Section 2.1.2 provides a formal definition of the abstract program behavior.

## 2.1.1 Concrete Program Semantics

**Definition 2.1** (Control-Flow Graph, Path)
*A* control-flow graph *(CFG) is a directed graph* $G = (V, v_s, E)$*, with a set of vertices* $V$*, a unique vertex* $v_s \in V$*, and a set of edges* $E \subset V \times V$*. The vertex* $v_s$ *is the unique* control-flow start *such that there exists no edge* $e \in E$ *with* $e = (v, v_s)$ *for any* $v \in V$*. The vertices in a CFG represent the program statements. The edges in a CFG describe all possible control-flow transitions of a program.*

*A* path *through the CFG* $G$ *is a sequence of vertices* $\pi = (v_1, v_2, \ldots, v_n) \in V^*$ *with* $v_1 = v_s$ *and* $(v_i, v_{i+1}) \in E$ *for all* $1 \leq i < n$ *and* $n \geq 1$*. Each vertex in* $v \in V$ *of the CFG* $G$ *is required to be reachable, i.e., there is a* path $\pi$ to $v$ *such that* $\pi = (v_1, \ldots, v_n) \in V^*$ *and* $(v_n, v) \in E$*. The* path extension to $v$ *where* $\pi = (v_1, \ldots, v_n)$ *and* $(v_n, v) \in E$ *is written as* $\pi \circ v$*. The path* $\pi = \epsilon$ *is* the empty path.

**Definition 2.2** (Concrete Transformer)
*A* concrete transformer *is a function* $f : V \to \mathcal{D} \to \mathcal{D}$ *that computes the effect of program statements on concrete states* $\delta \in \mathcal{D}$*. For any* $v \in V$ *the function* $f(v) : \mathcal{D} \to \mathcal{D}$ *computes the modification of concrete states due to execution of the program statement* $v$*.*

Given a control-flow graph $G = (V, v_s, E)$ and a concrete transformer $f : V \to \mathcal{D} \to \mathcal{D}$, we can now formally define the semantics of a path through the CFG.

**Definition 2.3** (Path Semantics)
*Let $G = (V, v_s, E)$ be a control-flow graph, and $f : V \to \mathcal{D} \to \mathcal{D}$ be a concrete transformer. The* path semantics $[\![\pi]\!] : \mathcal{D} \to \mathcal{D}$ *is defined as:*

$$[\![\pi]\!] = \begin{cases} \text{id} & \text{if } \pi = \epsilon \\ f(v_n) \circ [\![\pi']\!] & \text{if } \pi = \pi' \circ v_n \text{ is a path} \end{cases}$$

Program analyses typically compute program properties for sets of initial states instead for a single initial state. For this purpose we first lift the concrete transformer to sets of states. The *collecting transformer* $f_{coll} : V \to 2^{\mathcal{D}} \to 2^{\mathcal{D}}$ is defined for any $v \in V$ and $\Delta \in 2^{\mathcal{D}}$ as follows:

$$f_{coll}(v)(\Delta) := \{ f(v)(\delta) \mid \delta \in \Delta \}$$

The tuple $(2^{\mathcal{D}}, \subseteq, \bigcup, \bigcap, \emptyset, \mathcal{D})$ forms a complete lattice. The partial order $\subseteq$ arranges elements in $2^{\mathcal{D}}$ according to their precision. A set of states $a$ is more precise than $b$, iff $a \subseteq b$, because $a$ contains fewer concrete states than $b$. Hence, the collecting transformer is monotone: more precise input leads to more precise output.

By means of the collecting transformer, we can now lift the path semantics to a *collecting path semantics* as follows.

**Definition 2.4** (Collecting Path Semantics)
*Let $G = (V, v_s, E)$ be a control-flow graph, and $f_{coll} : V \to 2^{\mathcal{D}} \to 2^{\mathcal{D}}$. We define the* collecting path semantics $[\![\pi]\!]_{coll} : 2^{\mathcal{D}} \to 2^{\mathcal{D}}$ *as:*

$$[\![\pi]\!]_{coll} = \begin{cases} \text{id} & \text{if } \pi = \epsilon \\ f_{coll}(v_n) \circ [\![\pi']\!]_{coll} & \text{if } \pi = \pi' \circ v_n \text{ is a path} \end{cases}$$

Given a CFG representation for a program, the collecting path semantics formally describes the transformation of a set of initial states to a set of final states that can ever occur after the execution of the program. For certain program analyses, such as the pipeline analysis, we also need to be able to compute intermediate properties, such as which memory areas are potentially being accessed by a memory reference. Hence, we define the *sticky collecting semantics* $[\![v]\!]_{coll} : V \to 2^{\mathcal{D}}$ that maps each program statement to a set of possible states that can occur after that statement.

**Definition 2.5** (Sticky Collecting Semantics)
*Let $G = (V, v_s, E)$ be a control-flow graph, and $I \in 2^{\mathcal{D}}$ be the set of initial states. The* sticky collecting semantics $[\![v]\!]_{coll} : V \to 2^{\mathcal{D}}$ *is then defined as:*

$$[\![v]\!]_{coll} = \bigcup \{ [\![\pi]\!]_{coll}(I) \mid \pi \text{ is a path to } v \}$$

Note that other program properties need different semantics. For instance, whether a cache block is a useful cache block (i.e., it is possibly accessed again later) cannot be expressed with sticky collecting semantics [2]. For our purposes the sticky collecting semantics suffices for modeling the execution behavior of a processor because the concrete hardware state at a program point depends on the execution history and not its future.

In general, the sticky collecting semantics for a program point is not computable. This is either caused by an infeasibly large set of initial states, or by an arbitrary number of paths reaching that program point. The following section provides a solution to this problem on the basis of abstraction, which allows us to compute an over-approximation of the sticky collecting semantics.

## 2.1.2 Abstract Program Semantics

As Section 2.1.1 shows, we cannot compute the sticky collecting semantics in general. Hence, we translate the problem from the sets of concrete states $\mathcal{D}$ to an abstract domain $\hat{\mathcal{D}}$ with the partial order $\sqsubseteq$.

Similarly to the concrete case, the partial order can be understood as *more precise than*. This means that for two elements $a, b \in \hat{\mathcal{D}}$ the statement $a \sqsubseteq b$ means $a$ denotes more precise information than $b$. The abstract domain $\hat{\mathcal{D}}$ should be a complete lattice $(\hat{\mathcal{D}}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ such that a least upper bound exists for subsets of $\hat{\mathcal{D}}$. For this domain we need a monotone *abstract transformer* $f_{abs} : V \to \hat{\mathcal{D}} \to \hat{\mathcal{D}}$ that computes the effect of program statements directly on abstract states.

Analogously to the concrete domain, we can now formally define the *abstract collecting path semantics* as follows.

**Definition 2.6** (Abstract Collecting Path Semantics)
*Let $G = (V, v_s, E)$ be a control-flow graph, and $f_{abs} : V \to \hat{\mathcal{D}} \to \hat{\mathcal{D}}$ be an abstract transformer. The* abstract collecting path semantics $[\![\pi]\!]_{abs} : \hat{\mathcal{D}} \to \hat{\mathcal{D}}$ *is defined as:*

$$[\![\pi]\!]_{abs} = \begin{cases} \mathrm{id} & \text{if } \pi = \epsilon \\ f_{abs}(v_n) \circ [\![\pi']\!]_{abs} & \text{if } \pi = \pi' \circ v_n \text{ is a path} \end{cases}$$

We need to relate the abstract with the concrete domain to argue about the soundness of abstract semantics with respect to the concrete collecting semantics. For this purpose we require a monotone *concretization function* $\gamma : \hat{\mathcal{D}} \to 2^{\mathcal{D}}$ mapping an abstract state to a set of concrete states. The monotonicity of $\gamma$ guarantees that the partial order $\sqsubseteq$ on $\hat{\mathcal{D}}$ arranges abstract states according to their precision. So for any $a, b \in \hat{\mathcal{D}}$ with $a \sqsubseteq b$ it holds $\gamma(a) \subseteq \gamma(b)$. Clearly, the abstract state $a$ is more precise than the abstract state $b$ because it describes fewer concrete states. On this basis we

now formally define the *local consistency* of the abstract transformer $f_{abs}$ with the collecting transformer $f_{coll}$.

**Definition 2.7** (Local Consistency)
*Let $G = (V, v_s, E)$ be a CFG, $\gamma : \hat{\mathcal{D}} \to 2^{\mathcal{D}}$ be a concretization function. An abstract transformer $f_{abs}$ is* locally consistent *with a collecting transformer $f_{coll}$, iff:*

$$\forall v \in V, \hat{\delta} \in \hat{\mathcal{D}} : f_{coll}(v)(\gamma(\hat{\delta})) \subseteq \gamma(f_{abs}(v)(\hat{\delta}))$$

To ensure the soundness, the abstract transformer $f_{abs}$ should compute an over-approximation of the collecting semantics $f_{coll}$.

**Lemma 2.1** (Soundness of Abstract Collecting Path Semantics)
*The abstract collecting path semantics is a sound over-approximation of the collecting path semantics, i.e., for all $\hat{\delta} \in \hat{\mathcal{D}}$ it holds $([\![\pi]\!]_{coll} \circ \gamma)(\hat{\delta}) \subseteq (\gamma \circ [\![\pi]\!]_{abs})(\hat{\delta})$, if $f_{abs}$ is locally consistent with $f_{coll}$.*

*Proof.* Proof by structural induction over the path $\pi$. For the empty path, the claim is obviously true. For the induction step, we need to show that for any $\hat{\delta} \in \hat{\mathcal{D}}$ it holds $([\![\pi \circ v_n]\!]_{coll} \circ \gamma)(\hat{\delta}) \subseteq (\gamma \circ [\![\pi \circ v_n]\!]_{abs})(\hat{\delta})$.

Let $\hat{\delta} \in \hat{\mathcal{D}}$ be an abstract state:

$$\begin{aligned}
([\![\pi \circ v_n]\!]_{coll} \circ \gamma)(\hat{\delta}) &= (f_{coll}(v_n) \circ [\![\pi]\!]_{coll} \circ \gamma)(\hat{\delta}) && \mid \text{Definition} \\
&\subseteq (f_{coll}(v_n) \circ \gamma \circ [\![\pi]\!]_{abs})(\hat{\delta}) && \mid \text{Induction hypothesis} \\
&\subseteq (\gamma \circ f_{abs}(v_n) \circ [\![\pi]\!]_{abs})(\hat{\delta}) && \mid \text{Local consistency} \\
&= (\gamma \circ [\![\pi \circ v_n]\!]_{abs})(\hat{\delta}) && \mid \text{Definition}
\end{aligned}$$

□

Opposed to the concretization function, we can define a monotone *abstraction function* $\alpha : 2^{\mathcal{D}} \to \hat{\mathcal{D}}$ that computes for a set of concrete states $\Delta \in 2^{\mathcal{D}}$ the (best) corresponding abstract state $\hat{\delta} \in \hat{\mathcal{D}}$. The functions $\alpha$ and $\gamma$ should be *strongly adjoint*.

**Definition 2.8** (Strongly Adjoint)
*Let $(2^{\mathcal{D}}, \subseteq)$ and $(\hat{\mathcal{D}}, \sqsubseteq)$ be partially ordered sets, $\alpha : 2^{\mathcal{D}} \to \hat{\mathcal{D}}$ an abstraction function, and $\gamma : \hat{\mathcal{D}} \to 2^{\mathcal{D}}$ a concretization function. The functions $\alpha$ and $\gamma$ are* strongly adjoint, *iff for all $\Delta \in 2^{\mathcal{D}}$ and $\hat{\delta} \in \hat{\mathcal{D}}$:*

$$\Delta \subseteq \gamma(\alpha(\Delta))$$
$$\hat{\delta} = \alpha(\gamma(\hat{\delta}))$$

The first condition is required for soundness. It implies that after abstraction and concretization we can only lose information. This is unavoidable because the abstract domain cannot distinguish certain concrete states due to the abstraction. The second condition avoids that two different abstract states represent the same concrete states, which would be undesirable.

**Definition 2.9** (Abstract Sticky Collecting Semantics)
*Let $G = (V, v_s, E)$ be a control-flow graph, and $\hat{I} \in \hat{\mathcal{D}}$ be an initial abstract state. We define the* abstract sticky collecting semantics $[\![v]\!]_{abs} : V \to \hat{\mathcal{D}}$ *as:*

$$[\![v]\!]_{abs} = \bigsqcup \left\{ [\![\pi]\!]_{abs}(\hat{I}) \mid \pi \text{ is a path to } v \right\}$$

**Theorem 2.1** (Soundness of Abstract Sticky Collecting Semantics)
*The abstract sticky collecting semantics is a sound over-approximation of the sticky collecting semantics, i.e., for all $v \in V$ it holds $[\![v]\!]_{coll} \subseteq \gamma([\![v]\!]_{abs})$ if $f_{abs}$ is locally consistent with $f_{coll}$ and the abstract initial state $\hat{I}$ is $\hat{I} = \alpha(I)$ for the set of initial states $I \in 2^{\mathcal{D}}$.*

*Proof.* Let $v \in V$ be an arbitrary program statement:

$$\begin{aligned}
[\![v]\!]_{coll} &= \bigcup \{ [\![\pi]\!]_{coll}(I) \mid \pi \text{ is a path to } v \} && \mid \text{Definition} \\
&\subseteq \bigcup \{ [\![\pi]\!]_{coll}(\gamma(\alpha(I))) \mid \pi \text{ is a path to } v \} && \mid \alpha, \gamma \text{ strongly adjoint} \\
&\subseteq \bigcup \left\{ \gamma([\![\pi]\!]_{abs}(\hat{I})) \mid \pi \text{ is a path to } v \right\} && \mid \text{Lemma 2.1, definition} \\
&\subseteq \gamma\left( \bigsqcup \left\{ [\![\pi]\!]_{abs}(\hat{I}) \mid \pi \text{ is a path to } v \right\} \right) && \mid \text{Monotonicity of } \gamma \\
&= \gamma([\![v]\!]_{abs}) && \mid \text{Definition}
\end{aligned}$$

□

Figure 2.1 depicts the over-approximation of concrete program semantics by means of abstraction. The abstract transformer may produce abstract states describing concrete states that would not occur in reality. This loss of precision is an inherent property of abstraction and cannot be avoided.

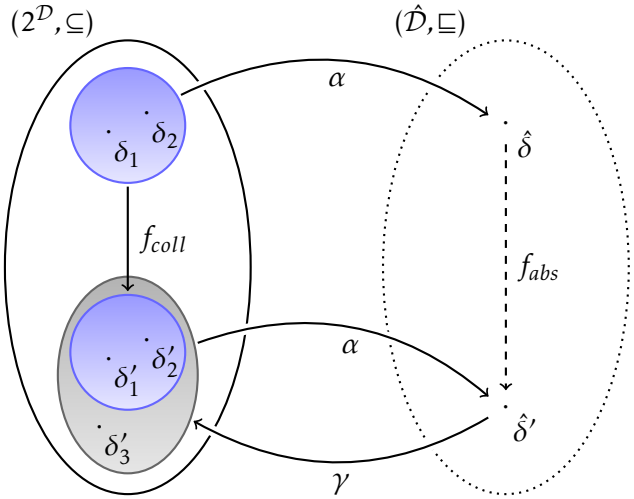



Figure 2.1: *Abstract Interpretation Principle:* Conclusions about the concrete program behavior are drawn out of abstract states computed by the abstract transformer. The gained information cannot be arbitrarily precise. Here the abstract state $\hat{\delta}'$ over-approximates the set of concrete states that actually can occur.

# Timing Analysis 3

Timing analysis is a major component of the verification process of a safety-critical software system. Safe and precise best- and worst-case execution time bounds for each of the system tasks are required to check whether the overall system is able to perform as required.

Section 3.1 discusses dynamic and static approaches to compute worst-case execution time bounds for programs. Again we provide evidence why measurement-based timing analysis approaches are usually unable to derive safety guarantees. Finally, we introduce the structure of state-of-the-art static timing analyzers and briefly discuss the used analysis techniques. Section 3.2 discusses the static architectural analysis in more detail. We investigate the types of imprecision that arise during the abstract simulation of the hardware state and relate them to corresponding hardware features, where possible. We conclude this chapter by formally describing the functioning of a static architectural analysis and provide the necessary means to define (and detect) timing anomalies or domino effects.

## 3.1 Overview

Exact worst-case execution times are often impossible or very hard to determine, even for the restricted class of safety-critical software systems. The reasons for this situation are manifold:

- The complex software structure accounts for this problem. Many control-flow decisions are based on input data and are thus impossible to compute statically. Certain paths through the program might be infeasible depending on the current state of the environment. Different modes of operation are typically found in safety-critical software systems.

  For example, a flight control system might behave differently if the plane is *on ground* or *in the air* and may thus exhibit a different timing behavior. A monolithic worst-case execution time analysis that does not discern between operating modes might not be precise enough. Instead one should discern between the modes of operation of the software [23].

- As introduced in Section 1.1 on page 2 the hardware architecture greatly influences the execution time behavior. The number of (initial) hardware states that lead to a different (overall) timing behavior of the processor is often directly related to the available (average-case) performance-enhancing features. Based on this observation, we consider the number of concrete hardware states

leading to a different timing behavior as a metric for the *complexity* of the processor.[5]

For example, consider a processor that is connected to a memory module with a read buffer storing the last access. Neglecting the structure of the processor pipeline, we can identify two states that lead to a different timing behavior: Either the next memory access hits the read buffer, or it does not. If we now add a cache in between the processor and the buffered memory, we observe an additional level of complexity: The next memory access either hits the cache, it hits the memory read buffer, or it misses both. Section 3.2 on page 30 discusses this issue in more detail.

WCET analyzers typically only provide worst-case timing guarantees, which are safe and precise upper bounds for the execution times of tasks. A timing analysis should exhibit the following properties:

**soundness** to ensure the reliability of the derived guarantees.

**precision** such that given timing constraints can be proven.

**efficiency** to make the analysis feasible in industrial practice.

Section 3.1.1 and Section 3.1.2 discuss two existing approaches to solve the timing analysis problem.

### 3.1.1 Dynamic Analysis Methods

Measurement-based WCET analyzers attempt to determine worst-case execution time bounds by means of repetitive execution of the analyzed program under varying conditions. This usually also requires instrumentation of the program code that is subject to timing analysis. The (additional) execution of the instrumentation code can lead to a non-negligible impact on the program's timing behavior, e.g., due to cache pollution, or modification of hardware buffers. The required amount of measured data drastically increases with increasing complexity of the program control flow.

Certification standards, such as DO-178B, (only) require that the software has been tested in accordance with the modified condition/decision coverage (MC/DC) criterion. The measurements as a whole have to meet the following requirements [11]:

- Each decision accepts every possible value.
- Each condition in a decision takes on every possible outcome.

[5] This metric does not say how complex it is to analyze a certain hardware feature, i.e., how precisely an analysis can predict its (timing) behavior.

- Each entry and exit point is invoked.
- Each condition (i.e., atomic Boolean expression) in a decision (i.e., composition of conditions) is shown to independently affect the outcome of the decision. Independence of a condition is shown by proving that only one condition changes at a time.

MC/DC-based measurements are typically unable to notice every possible evolution of hardware states that could occur during program execution. To ensure soundness, the measurements would additionally need to observe all feasible input scenarios, including all possible initial hardware states.

An extreme example (demonstrated in [14]) for a function with good average-case performance and bad WCET predictability is the library function `lDivMod` of the CodeWarrior V4.6 compiler for the Freescale HCS12X processor. The purpose of this routine is to compute quotient and remainder of two 32 bit unsigned integers. The algorithm performs an iteration computing successive approximations to the final result. To get an impression on the number of loop iterations, we performed an experiment in which `lDivMod` was applied to $10^8$ random inputs.

| Iteration Counts | Occurrence | Observed for |
|---|---|---|
| 0 | 1 552 | |
| 1 | 99 881 801 | |
| 2 | 116 421 | |
| 3 | 114 | |
| 4 .. 9 | 13 | |
| 10 .. 19 | 19 | |
| 20 .. 39 | 24 | |
| 40 .. 59 | 22 | |
| 60 .. 79 | 13 | |
| 80 .. 99 | 11 | |
| 100 .. 135 | 7 | |
| 156 | 1 | `lDivMod(0xffd93580, 0x107d228)` |
| 186 | 1 | `lDivMod(0xfff2c009, 0x118dcc4)` |
| 204 | 1 | `lDivMod(0xffe870e3, 0x1414167)` |

Table 3.1: *Observed iteration counts for `lDivMod`:* For the most inputs we observed a single iteration of the algorithm. Very rarely we found inputs to the algorithm causing a very high number of iterations.

Table 3.1 shows which iteration counts were observed in this experiment. The number of iterations is 1 in more than 99.8% and 0, 1, or 2 in more than 99.999%

of the sample inputs. On the other hand, iteration counts of more than 150 could be observed for a few specific inputs. There seems to be no simple way to derive the number of iterations from given inputs (other than running the algorithm). If a dynamic worst-case execution time analysis method only observes the fast iterations, it could possibly underestimate the overall execution time.

## 3.1.2 Static Analysis Methods

A static WCET analyzer determines a worst-case execution time bound of a code snippet (program task) in several analysis phases, as shown in Figure 3.1.

Opposing to dynamic WCET analysis methods, static WCET analyses do not actually execute the analyzed program on the real hardware. Furthermore this method does not require any modification of the program code. However, some abstraction of the execution platform is necessary to make a timing analysis of a complex software system and hardware architecture feasible. It is unavoidable that the abstraction incurs a loss of information about the hardware state.

How much information is lost depends on the methods used for timing analysis, on system properties, such as the hardware architecture, and on the analyzability and (static) predictability of the software. Virtually it is not possible to exactly state the degree of overestimation because the precise worst-case execution time is not measurable. Despite the potential loss of precision caused by abstraction, static timing analysis methods are well established in the industrial process.

The different phases of static WCET bound computation are briefly discussed in the following.

### Decoding Phase

In this phase the tool processes the input (binary) program. The decoder identifies the machine instructions and reconstructs the control-flow graph. Here, the user may provide additional information that is passed to each analysis phase. Such information could be targets of computed calls (used during the decoding phase), the number of iterations for a specific loop (used during the loop and value analysis phase), the hardware configuration (as required by the architectural analysis), and flow constraints (used in the path analysis phase).

### Loop and Value Analysis

The loop analysis phase tries to automatically compute upper bounds of loop iterations for all loops. The user may provide loop bounds if upper bounds cannot be determined. Additionally, the user may refine automatically computed loop bounds.

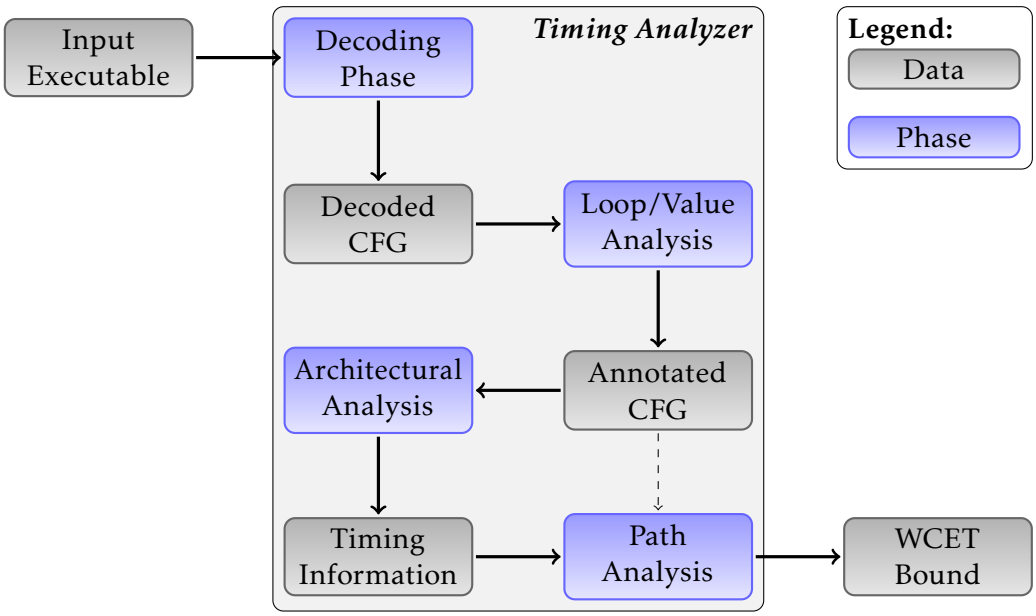



Figure 3.1: *Phases of WCET Computation:* Static timing analysis is split up into four phases. The first analysis phase reconstructs the control-flow graph from the input binary. The loop and value analysis derives the number of loop iterations and determines the potential targets of memory accesses. By means of abstract program simulation the architectural analysis computes timing information. The path analysis phase determines the worst-case execution path and an upper bound for the WCET.

The value analysis determines safe approximations of the values of processor registers and memory cells for every program point and execution context. Contents of registers or memory cells as well as address ranges for memory accesses may be provided by user annotations.

Loop and value analysis (and every analysis thereafter) distinguish contexts for a program point by means of *call strings* [39], which indicate part of the call stack leading to the routine containing the program point. The call string length is usually limited to reduce the number of contexts so that an efficient analysis is feasible for complex programs. Larger call string lengths usually lead to better value analysis results. In this fashion the sticky collecting semantics (see Definition 2.5 on page 18) is extended to a *context-sensitive* sticky collecting semantics, which is able to discern between different execution contexts for a program statement.

For example, the call string $\texttt{Proc1:0x188c} \rightarrow \texttt{Proc3}$ describes a context in which the routine `Proc3` has been called from routine `Proc1` at address `0x188c`.

### Architectural Analysis

Processing the annotated control-flow graph, the architectural analysis simulates the execution behavior of the input program through an abstract hardware model. The analysis determines lower and upper bounds for the execution times of basic blocks by performing an abstract interpretation of the program execution on the particular architecture, taking into account its pipeline, caches, memory buses, and attached peripheral devices [9, 42, 13].

Typically, the architectural analysis is a composition of a pipeline analysis and a cache analysis. By means of an abstract model of the hardware architecture, the pipeline analysis simulates the execution of each instruction. The cache analysis provides safe approximations of the contents of the caches at each program point. Complex architectural features are the main challenges for this analysis phase.

Since most parts of the pipeline state influence timing, current abstract models closely resemble the concrete hardware. The more performance-enhancing features a pipeline has, the larger is the search space. Superscalar- and out-of-order execution increase the number of possible interleavings. The larger the buffers (e.g., fetch buffers, retirement queues, etc.), the longer the influence of past events last. Dynamic branch prediction, speculative execution, cache-like structures, and branch history tables increase history dependence even more.

All these features influence execution time. To compute a precise bound on the execution time of a basic block, the analysis needs to exclude as many *timing accidents* as possible. Such accidents are data hazards, branch mis-predictions, occupied functional units, full queues, for example.

Abstract states may lack information about the state of some processor components, e.g., caches, queues, or predictors. Transitions of the pipeline may depend on such missing information. This causes the abstract pipeline model to become non-deterministic although the concrete pipeline is deterministic. When dealing with this non-determinism, one could be tempted to design the WCET analysis such that only the most expensive pipeline transition is chosen. However, in the presence of timing anomalies [24, 34] this approach is unsound. Thus, in general, the analysis has to follow all possible successor states.

### Path Analysis

Using the results of the preceding loop and value analysis and the architectural analysis phases, the path analysis phase estimates the worst-case execution path and computes a safe estimate for the WCET. The analysis translates the control-flow graph with the basic block timing bounds determined by the architectural analysis and the loop (and recursion) bounds derived by the loop and value analysis phase into an integer linear program [41]. The solution of the ILP yields the worst-case

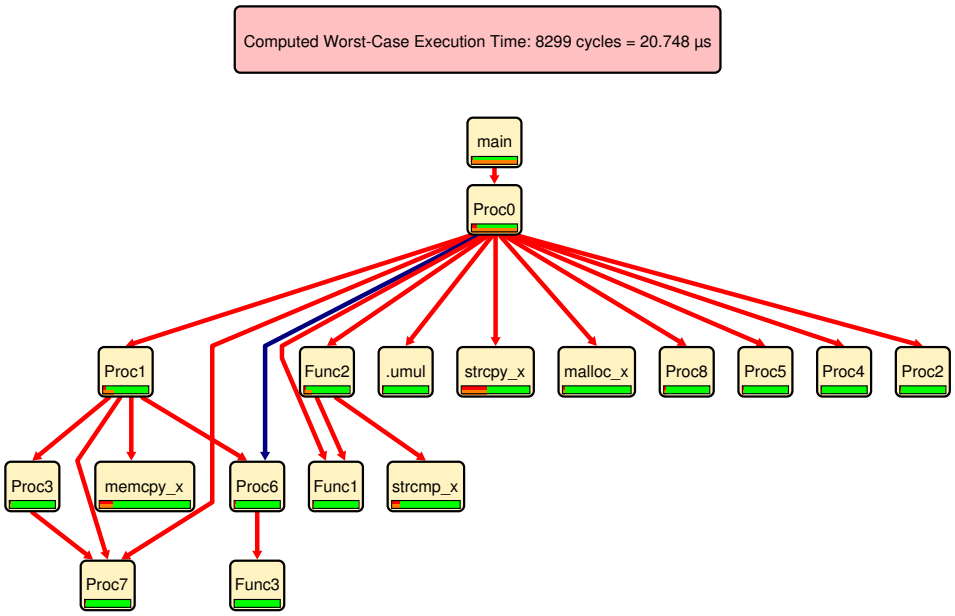



Figure 3.2: *WCET-Computation Result:* The result of the computation is a WCET-annotated call graph. Red edges depict the estimated execution path that triggers the worst-case timing behavior. Blue edges indicate program paths that do not contribute to the worst-case.

path and a safe approximation of the worst-case execution time. Figure 3.2 shows an example.

However the WCET bound computation based on ILP introduces an additional degree of overestimation. An ILP-based path analysis approach operates on the control-flow graph level and does forget information that is available during the architectural analysis. Such a path analysis method only uses upper bounds for the execution time of control-flow graph nodes (typically basic blocks) to determine a longest path through the analyzed program. Hence, any information about the abstract simulation across control-flow graph node boundaries is lost.

Figure 3.3 provides an example. For both basic blocks the upper bound for their execution is four cycles. Hence, solving the ILP yields eight cycles as an upper bound for their execution. This means that the ILP has assumed an evolution of processor states that does not occur in the abstract state space. Being aware of the cycle-wise state evolution an alternative path analysis approach would compute six cycles as an upper bound.

The *prediction graph* comprises the necessary information. Intuitively, the prediction graph is a description of all events (e.g., a cache hit, an access to a specific memory

module, etc.) that could occur on the actual hardware during the execution of the input program.

In Section 3.2 we formally introduce the prediction graph as an alternative to the ILP-based path analysis approach. Chapter 4 on page 51 defines the notions of timing anomaly and of domino effect by means of the prediction graph.

The prediction graph can also be used to validate the correctness of an abstraction and hence of a static (architectural) analysis, as discussed in Chapter 5 on page 63. For example, if we can show that the prediction graph for a specific program contains a measured sequence of events, we have evidence that the analysis works correctly for the hardware state before the start of the measurement. If we encounter a sequence of events that is not part of the prediction graph, the abstract hardware model is obviously invalid. Wrong abstractions can arise from invalid hardware specifications.

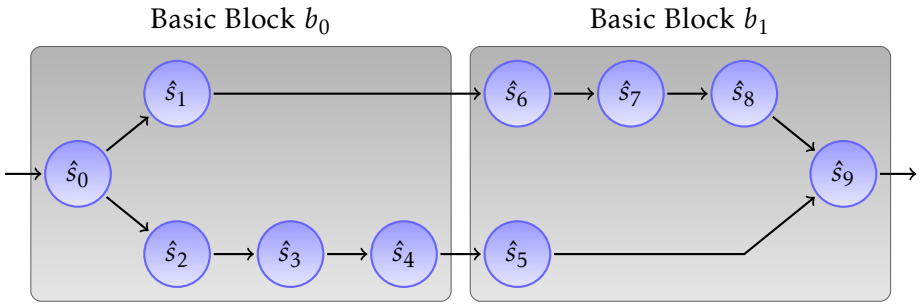



Figure 3.3: *ILP Path Analysis Problem:* The nodes represent abstract processor states. The edges denote the single cycle transitions in the abstract state space during the architectural analysis. This information is unavailable to ILP-based path analysis methods. Their input are solely upper bounds for the execution times of basic blocks. Hence, an ILP-based approach computes eight cycles as an upper bound. However, any execution of the basic blocks $b_0$ and $b_1$ already terminates after six cycles.

## 3.2 Architectural Analysis

Before we can argue about the correctness of a static architectural analysis, we first need to define the concrete behavior of a hardware architecture. For this purpose we require a behavioral hardware model that describes the processor's timing behavior on a per-action basis. This high level of granularity is required to be able to investigate which processor feature causes a timing anomaly (see Chapter 4 on page 51). Only observing the execution behavior on the instruction level, we would not be able to precisely identify the source of a timing anomaly. A modern

processor typically executes instructions out-of-order and may thus perform several actions at a time. For example, the processor requests the next instruction from memory while an arithmetic instruction completes execution. Hence, we require a specification that allows inspection of the processor-internal (parallel) processes.

Thesing [42] suggests that hardware representations on the level of a register transfer language (RTL) provide the necessary means. The use of RTL-level descriptions, such as VHDL [16] or Verilog [43], is beneficial because the hardware description provides the necessary insights to inspect the processor behavior on a clock cycle level.

VHDL or Verilog models are unfortunately hard to obtain because they are usually considered as (highly) confidential information. Yet, for some processors, such as the ERC32 and the LEON2 [10, 1], VHDL models have been made available to the public.

If a specification of the processor's behavior is unavailable, a hardware model has to be constructed from the available processor documentation. Furthermore, this model has to be refined in accordance with experiments on the real system. Chapter 6 demonstrates that each processor feature has to be thoroughly tested because the processor documentation sometimes does not match with the actual implementation.

In this section, we do not provide in-depth details on the modeling of a whole processor. For further reading we kindly point the interested reader to [42]. Section 3.2.1 formally introduces *concrete program simulation* by means of finite state automata. In Section 3.2.2 we describe *abstract program simulation* by abstraction from a concrete hardware model. Section 3.2.3 introduces the prediction graph based on which we are able to identify timing anomalies. Section 3.2.4 discusses the cause of non-determinism in abstract hardware simulation. Finally, Section 3.2.5 discusses challenges for static analysis.

### 3.2.1 Concrete Program Simulation

The actions that take place in a processor are triggered by signals on dedicated signal wires. For instance, the load-store unit posts a memory request on the bus by activating the transfer-start signal. Raising the corresponding transfer-acknowledge signal, the targeted memory module eventually answers the initial request. In most processor designs, these signals are synchronized by a clock signal oscillating between an active and an inactive state. In a *synchronous processor* design, the processor recognizes a signal shift upon the clock state change from inactive to active (i.e., rising clock edge) and from active to inactive (i.e., falling clock edge). This design allows us to argue that time in a computer is a *discrete* resource. The

time elapsing until a specific action takes place can then be expressed in terms of clock cycles (or half-clock cycles depending on the referenced clock edge). The execution of an instruction is typically synchronized with the rising clock edge.

However, there are successful attempts to implement clock-less processors using asynchronous circuits. The functional units of the processor wait for signals that indicate a certain kind of activity. An *asynchronous processor* design allows for higher performances with a significantly lower power consumption compared to a synchronous design [46]. To the best of our knowledge there is no (static) analysis approach available that copes with the increasing complexity caused by asynchronous logic.[6]

Thus, we restrict ourselves to the analysis of synchronous processor designs. The processor execution behavior is then modeled as the cycle-wise evolution of a finite state automaton. This approach is feasible because the processor and the connected memories, which are of bounded size, can only accept a finite number of states. We assume a (correct) model is available, e.g., by a semi-automatic derivation of a processor model from a VHDL description [37].

The execution behavior of a processor for a program is then simulated as follows: We select an initial (hardware) state where the program, encoded as sequence of instructions, is allocated to a dedicated place in memory. The initial state should also describe the situation where the processor is about to start executing the input program, i.e., the program counter is set up to the memory address of the very first instruction of that program. Then we follow the evolution of that initial state in accordance with the transitions in the finite automaton until the last instruction has completed execution and is about to leave the processor pipeline. Because each transition represents a single processor cycle, the total number of transitions represents the number of processor cycles the program takes to execute.

**Definition 3.1** (Finite State Automaton)
*A* finite state automaton *is a pair* $\mathcal{A} = (\mathcal{S}, \tau)$. *The set* $\mathcal{S}$ *denotes the finite set of states. The function* $\tau : \mathcal{S} \to \mathcal{S}$ *computes the cycle-wise transition from one state to another.*

*Any state* $s \in S$ *of the finite state automaton* $\mathcal{A}$ *comprises the program it executes. Let* $G = (V, v_s, E)$ *be a control-flow graph. If the program that corresponds to* $G$ *can be allocated to the processor memory, which is also part of the automaton, there exists a set of states* $\mathcal{S}_G \subseteq \mathcal{S}$ *that comprise the memory image of the program that is represented by* $G$.

To formally define the timing behavior of a processor we need to identify when an instruction has finished execution. The instruction set architecture (ISA) describes

[6] Complexity is increased due to Boolean logic being inadequate to describe the behavior of asynchronous circuits because signals are no longer guaranteed to have a discrete true or false state (i.e., active or inactive) at any given time.

the program semantics in a sequential way, but the processor may execute several instructions in parallel, of which some may leave the pipeline earlier. Thesing [42] solves this problem by means of state predicates to transcribe the parallel execution behavior into a sequential execution. For our purposes it suffices to introduce the *execution predicate* that denotes whether a state is about to execute an instruction.

**Definition 3.2** (Execution Predicate)
*Let $G = (V, v_s, E)$ be a CFG, $(\mathcal{S}, \tau)$ be a finite state automaton for a processor, $s \in \mathcal{S}$ a processor state, and $v \in V$ an instruction. If a state $s$ represents a processor state where the instruction $v$ is currently* under execution, *we denote this by $s \rhd v$. $s \rhd v \wedge \tau(s) \not\rhd v$ means that the instruction $v$ has* completed execution *and is about to leave the pipeline.*

To execute an instruction the processor performs a certain sequence of actions (or micro-operations) that depends on the type of instruction. Typical actions are, e.g., instruction fetch, decode, dispatch to the corresponding functional unit, memory access (if the instruction is a load-store instruction), and write-back of results. The notion of *action* is not limited to pipeline stages. Other actions that could occur during instruction execution are, e.g., cache-line fill, clock synchronization, and others. In this fashion we can mark state transitions with action markers to identify the starting and ending points of individual actions.

**Definition 3.3** (Action Marker)
*Let $(\mathcal{S}, \tau)$ be a finite state automaton, $s, s' \in \mathcal{S}$ states, $\Lambda$ be a non-empty alphabet of actions, and $\lambda \in \Lambda$ be an action. The state transition from $s$ to $\tau(s)$ is marked with an* action begin marker *if the action $\lambda$ is initiated, written $(s, \tau(s)) \vdash \lambda$. The state transition from $s'$ to $\tau(s')$ is marked with an* action end marker *if the action $\lambda$ completes that was initiated at state $s$, written $(s', \tau(s')) \dashv_s \lambda$.*

Figure 3.4 provides an example showing parts of the execution of a load-store instruction. Here we observe in detail the dispatch of the instruction to the corresponding function unit and the memory access phase.

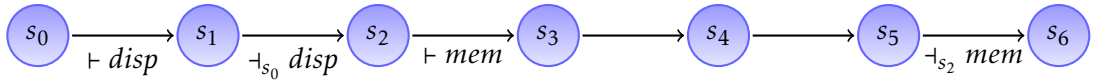



Figure 3.4: *Actions during Instruction Execution:* Possible sequence of actions that occurs during execution of a load-store instruction. In state $s_1$ the processor is about to dispatch the instruction (*disp* action). The load-store unit initiates a memory access in state $s_3$ (*mem* action). After three state transitions the result is finally available in state $s_6$.

By means of action markers we are later able to investigate the nature of timing anomalies. In Section 3.2.4 on page 43 we use action markers to identify local

worst-case decisions in the abstract state space.

In contrast to Thesing, we are not interested in the execution behavior on an instruction level. In order to investigate the source of timing anomalies, we need to be able to observe individual actions of the processor pipeline. Thus we extend the concrete transformer (see Definition 2.2 on page 17) to the *state transformer* as follows.

**Definition 3.4** (State Transformer)
*Let $G = (V, v_s, E)$ be a CFG, $(\mathcal{S}, \tau)$ be a finite state automaton for a processor. A* state transformer step $f_\tau^{\triangleright} : V \to \mathcal{S} \to \mathcal{S}$ *computes the effect of instructions $v \in V$ on concrete pipeline states $s \in \mathcal{S}$ on the state transition level.*

$$f_\tau^{\triangleright}(v)(s) := \begin{cases} \tau(s) & \text{if } s \triangleright v \ \wedge \tau(s) \triangleright v \\ s & \text{otherwise} \end{cases}$$

*Each occurrence of an instruction $v \in V$ has to* terminate *eventually. This means that for a state $s \in \mathcal{S}$ with $s \triangleright v$ there exists $n \in \mathbb{N}$ such that $s' = {f_\tau^{\triangleright}}^{n+1}(v)(s) = {f_\tau^{\triangleright}}^{n}(v)(s)$. The state $s'$ denotes the processor state in which $v$ has finished execution and is about to leave the pipeline. This additionally implies that each action has to terminate eventually, i.e., if $(s, \tau(s)) \vdash \lambda$ there exists $n \in \mathbb{N}$ such that $(\tau^n(s), \tau^{n+1}(s)) \dashv_s \lambda$.*

*A* state transformer $f_\tau : V \to \mathcal{S} \to \mathcal{S}$ *is a concrete transformer that computes the effect of instructions hiding the intermediate state transitions.*

$$f_\tau(v)(s) := {f_\tau^{\triangleright}}^{n}(v)(s) \quad \text{such that} \quad n \in \mathbb{N} \ \wedge \ {f_\tau^{\triangleright}}^{n+1}(v)(s) = {f_\tau^{\triangleright}}^{n}(v)(s)$$

The state transformer only updates the current processor state if the current instruction is still being executed. Otherwise, i.e., if the processor has just or previously finished the current instruction, we do not modify the current state. In this fashion we can describe the parallel execution of instructions in a sequential manner.

**Example** Consider a path $\pi = (v_1, v_2)$. Assume the processor is able to execute $v_2$ while executing $v_1$. Additionally, let us assume that $v_2$ completes earlier than the first instruction. Starting from an initial state $s$ there exists an $n \in \mathbb{N}$ such that $\tau^n(s) \triangleright v_1 \ \wedge \ \tau^n(s) \triangleright v_2 \ \wedge \ \tau^{n+1}(s) \not\triangleright v_2$. Eventually the processor will finish the execution of $v_1$. Hence, there is $m \in \mathbb{N}$ with $n < m \wedge \tau^m(s) \triangleright v_1 \wedge \tau^{m+1}(s) \not\triangleright v_1$. The execution time of $v_2$ is subsumed by the execution time of $v_1$. The state $\tau^m(s)$ is the state of the processor after the execution of the path $\pi$. The number of cycles to execute $\pi$ is hence $m$ processor cycles.

We first lift the state transformer to a *state collecting transformer* in order to define the *state collecting path semantics*. The state collecting transformer observes all

intermediate steps until an instruction $v$ finishes execution. For $v \in V$ and $\sigma \in 2^{\mathcal{S}}$ we define the *state collecting transformer* as follows:

$$f_{\tau_{coll}}(v)(\sigma) := \{ f_{\tau}(v)(s) \mid s \in \sigma \}$$

The tuple $(2^{\mathcal{S}}, \subseteq, \bigcup, \bigcap, \emptyset, \mathcal{S})$ forms a complete lattice. The partial order $\subseteq$ arranges elements in $2^{\mathcal{S}}$ according to their precision. A set of processor states $a$ is more precise than $b$ iff $a \subseteq b$ because $b$ possibly contains additional concrete processor states. The state collecting transformer is monotone: more precise input leads to more precise output.

The state collecting transformer allows us to define the state collecting path semantics similar to the collecting path semantics (see Definition 2.4).

**Definition 3.5** (State Collecting Path Semantics)
*Let $G = (V, v_s, E)$ be a CFG and $f_{\tau_{coll}} : V \to \mathcal{S} \to \mathcal{S}$ be a state collecting transformer. The* state collecting path semantics $[\![\pi]\!]_{\tau_{coll}} : 2^{\mathcal{S}} \to 2^{\mathcal{S}}$ *is defined as:*

$$[\![\pi]\!]_{\tau_{coll}} = \begin{cases} \mathrm{id} & \text{if } \pi = \epsilon \\ f_{\tau_{coll}}(v_n) \circ [\![\pi']\!]_{\tau_{coll}} & \text{if } \pi = \pi' \circ v_n \text{ is a path} \end{cases}$$

For a set of initial states the state collecting path semantics computes the set of states the processor will attain after the execution of a given path. The instructions on the path are processed in program order even if the processor is able to complete some instructions out of order. The state collecting path semantics advances to the next instruction if each state in the computed set of states represents a processor state where the current instruction is no longer being executed.

The state collecting path semantics is not practically computable in general because the number of initial hardware states to consider is obstructively large. The following section solves this problem by means of abstracting from the concrete hardware states – similarly to the abstract program semantics (see Section 2.1.2 on page 19).

## 3.2.2 Abstract Program Simulation

In general, we cannot compute the state collecting path semantics due to the finite but infeasibly large number of states in the finite automaton. We thus translate the problem from the set of concrete states $\mathcal{S}$ to an abstract domain $\hat{\mathcal{S}}$ with the partial order $\sqsubseteq$.

We need to relate the abstract with the concrete domain to argue about the soundness of abstract state semantics with respect to the concrete state semantics. For this purpose we require a monotone *state concretization function* $\gamma : \hat{\mathcal{S}} \to 2^{\mathcal{S}}$ mapping

an abstract state to a set of concrete states. The monotonicity of $\gamma$ guarantees that the partial order $\sqsubseteq$ on $\hat{\mathcal{S}}$ arranges abstract states according to their precision. So for any $a, b \in \hat{\mathcal{S}}$ with $a \sqsubseteq b$ it holds $\gamma(a) \subseteq \gamma(b)$. The abstract state $a$ is more precise than the abstract state $b$ because it describes fewer concrete states. We require the state concretization function $\gamma$ to be both *execution-deterministic* and *action-deterministic* to be able to determine whether an instruction is about to finish execution in an abstract hardware state.

**Definition 3.6** (Execution-Deterministic)
*Let $G = (V, v_s, E)$ be a CFG. A state concretization function $\gamma : \hat{\mathcal{S}} \to 2^{\mathcal{S}}$ is* execution-deterministic *iff for any instruction $v \in V$ and any abstract state $\hat{s} \in \hat{\mathcal{S}}$ it holds that all corresponding concrete states either do execute $v$, written $\hat{s} \rhd v$, or do not execute $v$, written $\hat{s} \not\rhd v$. This yields $\hat{s} \rhd v \Leftrightarrow \forall s \in \gamma(\hat{s}).s \rhd v$ and $\hat{s} \not\rhd v \Leftrightarrow \forall s \in \gamma(\hat{s}).s \not\rhd v$.*

**Definition 3.7** (Action-Deterministic)
*Let $G = (V, v_s, E)$ be a CFG and $\Lambda$ be a non-empty alphabet of actions. A state concretization function $\gamma : \hat{\mathcal{S}} \to 2^{\mathcal{S}}$ is* action-deterministic *iff for any states $\hat{s}, \hat{t}, \hat{v}, \hat{w} \in \hat{\mathcal{S}}$ and any action $\lambda \in \Lambda$ all corresponding concrete states are marked with the same action markers. This yields $(\hat{s}, \hat{t}) \vdash \lambda \Leftrightarrow \forall s \in \gamma(\hat{s}), t \in \gamma(\hat{t}).(s,t) \vdash \lambda$ and $(\hat{v}, \hat{w}) \dashv_{\hat{s}} \lambda \Leftrightarrow \forall v \in \gamma(\hat{v}), w \in \gamma(\hat{w}).\exists s \in \gamma(\hat{s}).(v,w) \dashv_s \lambda$.*

In this fashion we can determine for any abstract hardware state whether an instruction is being executed. This requirement also ensures that the abstract hardware model keeps track of the progression of the instruction through the processor pipeline. Otherwise the abstract state cannot deterministically decide whether an instruction is being executed or not. Consequently, we require the abstract state transition function to discern between abstract states that arise from disjoint assumptions (e.g., cache hit or cache miss).

Because we partition the abstract state space by requiring $\gamma$ to be execution- and action-deterministic, we lift the state concretization function $\gamma$ to $\overline{\gamma}$ that accepts sets of abstract states. The function $\overline{\gamma}$ retains the monotonicity of $\gamma$.

We define $\overline{\gamma} : 2^{\hat{\mathcal{S}}} \to 2^{\mathcal{S}}$, where $\overline{\gamma}(\hat{\sigma}) := \bigcup_{\hat{s} \in \hat{\sigma}} \gamma(\hat{s})$ for any $\hat{\sigma} \in 2^{\hat{\mathcal{S}}}$.

Opposed to the concretization function $\overline{\gamma}$, we can define a monotone *state abstraction function* $\overline{\alpha} : 2^{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$ that computes for a set of concrete states $\sigma \in 2^{\mathcal{S}}$ the (best) corresponding abstract state set in $2^{\hat{\mathcal{S}}}$. For soundness reasons the functions $\overline{\alpha}$ and $\overline{\gamma}$ should be strongly adjoint (see Definition 2.8 on page 20).

**Definition 3.8** (Abstract State Automaton)
*An* abstract state automaton *is a pair $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$. The set $\hat{\mathcal{S}}$ denotes the set of abstract states. The function $\tau_{abs} : \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}}}$ computes the cycle-wise evolution of abstract states.*

*The abstract automaton* corresponds *to the finite state automaton* $\mathcal{A} = (\mathcal{S}, \tau)$*, if there exist strongly adjoint state abstraction and state concretization functions* $\overline{\alpha}$ *and* $\overline{\gamma}$ *such that* $\gamma$ *is execution- and action-deterministic and* $\tau$ *and* $\tau_{abs}$ *are* locally consistent*, i.e., for all abstract states* $\hat{s} \in \hat{\mathcal{S}}$ *it holds* $\tau(\gamma(\hat{s})) \subseteq \gamma(\tau_{abs}(\hat{s}))$*. The abstract state transition function* $\tau_{abs}$ *computes a safe approximation of hardware state transitions that can occur in any concrete state transition.*

We use the abstract state automaton as a basis for the abstract simulation of the execution of a program. Due to the possible presence of timing anomalies and to entail the full system behavior the abstract state transition may compute several abstract successor states, i.e., $|\tau_{abs}(\hat{s})| > 1$ is possible for some $\hat{s} \in \hat{\mathcal{S}}$. This is called *split*. Section 3.2.4 on page 43 investigates splits in more detail.

Given the abstract state automaton we can now formally define the *abstract state transformer* as follows.

**Definition 3.9** (Abstract State Transformer)
*Let* $G = (V, v_s, E)$ *be a CFG,* $\gamma$ *an execution-deterministic state concretization function, and* $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ *be an abstract state automaton that corresponds to a finite state automaton.*

*An* abstract state transformer step $f^{\triangleright}_{\tau_{abs}} : V \to 2^{\hat{\mathcal{S}}} \to 2^{\hat{\mathcal{S}}}$ *computes the effect of instructions on the transition level.*

$$f^{\triangleright}_{\tau_{abs}}(v)(\hat{\sigma}) := \bigcup_{\hat{s} \in \hat{\sigma}} \{\hat{t} \mid \hat{s} \triangleright v \wedge \hat{t} \in \tau_{abs}(\hat{s}) \wedge \hat{t} \triangleright v\} \cup \{\hat{s} \mid \hat{s} \not\triangleright v \vee \exists \hat{t} \in \tau_{abs}(\hat{s}).\hat{t} \not\triangleright v\}$$

*For a given instruction* $v \in V$*, the abstract state transformer step eventually reaches a fixed point. Because* $\gamma$ *is execution-deterministic and every instruction terminates eventually, there exists an* $n \in \mathbb{N}$ *such that* $f^{\triangleright^n}_{\tau_{abs}}(v)(\hat{\sigma}) = f^{\triangleright^{n+1}}_{\tau_{abs}}(v)(\hat{\sigma})$*. Every abstract state* $\hat{t} \in f^{\triangleright^n}_{\tau_{abs}}(v)(\hat{\sigma})$ *then describes a situation where* $v$ *is about to leave the (abstract model of the) processor pipeline.*

*An* abstract state transformer $f_{\tau_{abs}} : V \to 2^{\hat{\mathcal{S}}} \to 2^{\hat{\mathcal{S}}}$ *computes the effect of instructions hiding the intermediate abstract state transitions.*

$$f_{\tau_{abs}}(v)(\hat{\sigma}) := f^{\triangleright^n}_{\tau_{abs}}(v)(\hat{\sigma}) \quad \textit{such that} \quad n \in \mathbb{N} \wedge f^{\triangleright^{n+1}}_{\tau_{abs}}(v)(\hat{\sigma}) = f^{\triangleright^n}_{\tau_{abs}}(v)(\hat{\sigma})$$

The abstract state transformer continues to follow abstract state transitions until a set of abstract states is computed where each of the states is about to finish the execution of the occurrence of the instruction $v$.

Before we formally define *abstract state collecting path semantics*, we show that the abstract state transformer $f_{\tau_{abs}}$ is locally consistent with the state collecting transformer $f_{\tau_{coll}}$ if $\gamma$ is execution-deterministic and $\tau$ and $\tau_{abs}$ are locally consistent.

**Lemma 3.1** (Local Consistency of Abstract State Transformer)
*An abstract state transformer $f_{\tau_{abs}} : V \to 2^{\hat{\mathcal{S}}} \to 2^{\hat{\mathcal{S}}}$ is locally consistent with the state collecting transformer $f_{\tau_{coll}} : V \to \mathcal{S} \to \mathcal{S}$ if $\gamma$ is execution-deterministic and $\tau$ and $\tau_{abs}$ are locally consistent.*

*Proof.* Let $\hat{A} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton that corresponds to the finite state automaton $A = (\mathcal{S}, \tau)$ and $\gamma : \hat{\mathcal{S}} \to 2^{\mathcal{S}}$ be execution-deterministic. We prove that $f^{\triangleright}_{\tau_{abs}}$ is locally consistent with $f^{\triangleright}_{\tau}$. Let $G = (V, v_s, E)$ be a CFG, $v \in V$ be an instruction and $\hat{\sigma} \in 2^{\hat{\mathcal{S}}}$ be a set of abstract states.

$$
\begin{aligned}
& \bigcup_{\hat{s} \in \hat{\sigma}} \{ f^{\triangleright}_{\tau}(v)(s) \mid s \in \gamma(\hat{s}) \} \\
= & \bigcup_{\hat{s} \in \hat{\sigma}} \begin{array}{l} \{ s \mid s \in \gamma(\hat{s}) \wedge (s \not\rhd v \vee (s \rhd v \wedge \tau(s) \not\rhd v)) \} \\ \cup \ \{ t \mid s \in \gamma(\hat{s}) \wedge t = \tau(s) \wedge s \rhd v \wedge t \rhd v \} \end{array} && \mid \text{Definition} \\
\subseteq & \bigcup_{\hat{s} \in \hat{\sigma}} \begin{array}{l} \{ s \mid s \in \gamma(\hat{s}) \wedge (\hat{s} \not\rhd v \vee (\hat{s} \rhd v \wedge \tau(s) \not\rhd v)) \} \\ \cup \ \{ t \mid s \in \gamma(\hat{s}) \wedge t = \tau(s) \wedge \hat{s} \rhd v \wedge t \rhd v \} \end{array} && \mid \gamma \text{ execution determ.} \\
\subseteq & \bigcup_{\hat{s} \in \hat{\sigma}} \begin{array}{l} \{ \gamma(\hat{s}) \mid \hat{s} \not\rhd v \vee (\hat{s} \rhd v \wedge \exists \hat{t} \in \tau_{abs}(\hat{s}).\hat{t} \not\rhd v)) \} \\ \cup \ \{ \gamma(\hat{t}) \mid \hat{t} \in \tau_{abs}(s) \wedge \hat{s} \rhd v \wedge \hat{t} \rhd v \} \end{array} && \mid \tau, \tau_{abs} \text{ locally cons.} \\
\subseteq & \overline{\gamma} \left( \bigcup_{\hat{s} \in \hat{\sigma}} \begin{array}{l} \{ \hat{s} \mid \hat{s} \not\rhd v \vee (\hat{s} \rhd v \wedge \exists \hat{t} \in \tau_{abs}(\hat{s}).\hat{t} \not\rhd v)) \} \\ \cup \ \{ \hat{t} \mid \hat{t} \in \tau_{abs}(s) \wedge \hat{s} \rhd v \wedge \hat{t} \rhd v \} \end{array} \right) && \mid \text{Monotonicy of } \overline{\gamma} \\
= & \overline{\gamma} \left( f^{\triangleright}_{\tau_{abs}}(v)(\hat{\sigma}) \right) && \mid \text{Definition}
\end{aligned}
$$

The local consistency between $f_{\tau_{abs}}$ and $f_{\tau_{coll}}$ follows by iterating this property. □

By means of the abstract state transformer, we can then formally define the *abstract state collecting path semantics*.

**Definition 3.10** (Abstract State Collecting Path Semantics)
*Let $G = (V, v_s, E)$ be a CFG and $f_{\tau_{abs}} : V \to 2^{\hat{\mathcal{S}}} \to 2^{\hat{\mathcal{S}}}$ be an abstract state transformer. The* abstract state collecting path semantics $[\![\pi]\!]_{\tau_{abs}} : 2^{\hat{\mathcal{S}}} \to 2^{\hat{\mathcal{S}}}$ *is defined as:*

$$
[\![\pi]\!]_{\tau_{abs}} = \begin{cases} \mathrm{id} & \text{if } \pi = \epsilon \\ f_{\tau_{abs}}(v_n) \circ [\![\pi']\!]_{\tau_{abs}} & \text{if } \pi = \pi' \circ v_n \text{ is a path} \end{cases}
$$

The abstract state collecting path semantics computes the evolution of abstract states during the execution of a path through a program. Considering all paths through a program, the combined evolution of the abstract hardware states can be understood as a graph that contains all possible hardware decisions that can occur according to

the abstract state automaton. Later we use this representation to determine whether the execution of a program exhibits a timing anomaly.

But first we argue about the soundness of the abstract state collecting path semantics.

**Lemma 3.2** (Soundness of Abstract State Collecting Path Semantics)
*The abstract state collecting path semantics is a sound over-approximation of the state collecting path semantics, i.e., for all* $\hat{\sigma} \in 2^{\hat{S}}$ *it holds* $([\![\pi]\!]_{\tau_{coll}} \circ \overline{\gamma})(\hat{\sigma}) \subseteq (\overline{\gamma} \circ [\![\pi]\!]_{\tau_{abs}})(\hat{\sigma})$ *if* $\gamma$ *is execution-deterministic and* $\tau$ *and* $\tau_{abs}$ *are locally consistent.*

*Proof.* Proof by structural induction over the path $\pi$. Let $\hat{A} = (\hat{S}, \tau_{abs})$ be an abstract state automaton that corresponds to the finite state automaton $A = (S, \tau)$ and $\gamma : \hat{S} \to 2^{S}$ be execution-deterministic. For the empty path, the claim is obviously true. For the induction step, we need to show that for any $\hat{\sigma} \in 2^{\hat{S}}$ it holds $([\![\pi \circ v_n]\!]_{\tau_{coll}} \circ \overline{\gamma})(\hat{\sigma}) \subseteq (\overline{\gamma} \circ [\![\pi \circ v_n]\!]_{\tau_{abs}})(\hat{\sigma})$.

Let $\hat{\sigma} \in 2^{\hat{S}}$ be a set of abstract states:

$$
\begin{aligned}
([\![\pi \circ v_n]\!]_{\tau_{coll}} \circ \overline{\gamma})(\hat{\sigma}) &= (f_{\tau_{coll}}(v_n) \circ [\![\pi]\!]_{\tau_{coll}} \circ \overline{\gamma})(\hat{\sigma}) && \mid \text{Definition} \\
&\subseteq (f_{\tau_{coll}}(v_n) \circ \overline{\gamma} \circ [\![\pi]\!]_{\tau_{abs}})(\hat{\sigma}) && \mid \text{Induction hypothesis} \\
&\subseteq (\overline{\gamma} \circ f_{\tau_{abs}}(v_n) \circ [\![\pi]\!]_{\tau_{abs}})(\hat{\sigma}) && \mid \text{Lemma 3.1} \\
&= (\overline{\gamma} \circ [\![\pi \circ v_n]\!]_{\tau_{abs}})(\hat{\sigma}) && \mid \text{Definition}
\end{aligned}
$$

□

## 3.2.3 Prediction Graph

The abstract state collecting path semantics computes a safe approximation of the concrete execution behavior of a program. We can now combine the observed abstract state transitions into a *prediction graph* that describes every possible execution behavior. The prediction graph is our basis to detect instances of timing anomalies in the abstract hardware state space. For this purpose we define the *feasible abstract successor* to construct the prediction graph.

**Definition 3.11** (Feasible Abstract Successor)
*Let* $G = (V, v_s, E)$ *be a CFG and* $\hat{A} = (\hat{S}, \tau_{abs})$ *an abstract state automaton, and* $\theta \in 2^{\hat{S}}$ *be the set of initial abstract states that are about to execute the first instruction of the control-flow graph G, i.e.,* $\forall \hat{s} \in \theta \,.\, \exists \hat{t} \in \hat{S}$ *s.t.* $\hat{t} \not\rhd v_s \wedge \hat{s} \in \tau_{abs}(\hat{t}) \wedge \hat{s} \rhd v_s$.

*The abstract state $\hat{t}$ is a* feasible abstract successor *of the abstract state $\hat{s}$, written $\hat{s} \rightsquigarrow_\pi \hat{t}$, iff there exists a path $\pi = \pi' \circ v \circ \pi''$ and $k \in \mathbb{N}$ such that:*

$$\hat{s} \in f_{\tau_{abs}}^{\triangleright k}(v)([\![\pi']\!]_{\tau_{abs}}(\theta)) \wedge \hat{t} \in f_{\tau_{abs}}^{\triangleright k+1}(v)([\![\pi']\!]_{\tau_{abs}}(\theta)) \wedge \hat{t} \in \tau_{abs}(\hat{s})$$

**Definition 3.12** (Prediction Graph)
*Let $G = (V, v_s, E)$ be a CFG, $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ an abstract state automaton, and $\theta \in 2^{\hat{\mathcal{S}}}$ be the set of initial abstract states. The corresponding* prediction graph *is $\hat{\mathcal{P}}_G = (\hat{\mathcal{S}}, \hat{\mathcal{E}})$, where $\hat{\mathcal{E}} = \{(\hat{s}, \hat{t}) \mid \hat{s} \rightsquigarrow_\pi \hat{t} \wedge \pi$ is a path through $G\}$.*

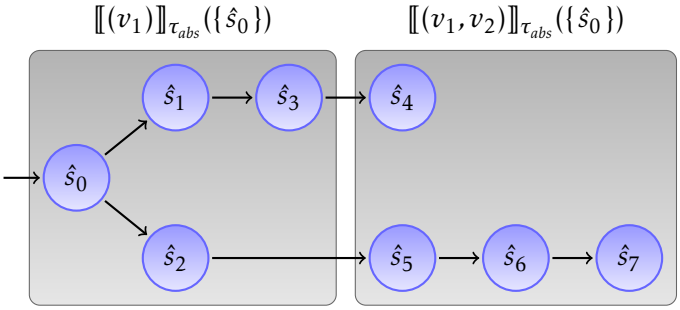



Figure 3.5: *Prediction Graph:* Evolution of abstract hardware states for the simulation of the instruction sequence $(v_1, v_2)$. The edges denote the single cycle transitions in the abstract state space. The gray boxes span the set of states that execute under the same instruction. In total the program completes after four cycles.

Figure 3.5 depicts the evolution of abstract hardware states for the execution of a very simple program that consists of the path $\pi = (v_1, v_2)$. The length of a longest path through the prediction graph provides a safe upper bound for the number of processor cycles in which the input program finishes execution. In this example, the longest path comprises four transitions of abstract hardware states. Hence, the program takes at worst four cycles to complete.

Depending on the control-flow graph, it is not feasible to construct the whole prediction graph. For now we restrict ourselves to programs that comprise a finite number of paths. Furthermore we implicitly require the abstract hardware states to carry along the analysis context. We assume an infinite call string approach and all loops to be virtually unrolled [26], such that every loop iteration corresponds to a different analysis context. In this fashion we achieve that the prediction graph is a directed-acyclic graph (DAG) because back edges cannot occur. Chapter 4 on page 51 relaxes some restrictions to allow for arbitrary, possibly infinite programs.

We can now compute a longest path through the prediction graph. The length of that path denotes an upper bound (in terms of processor cycles) for the longest execution

of the corresponding program. To do so we first sort the prediction graph nodes in topological order by means of depth-first search to realize the sorting. Algorithm 3.1 provides a pseudo-code implementation [44]. The output of the algorithm is a vector of topologically sorted nodes.

**Algorithm 3.1** Topological Sorting of the Prediction Graph

```
function TopologicalSorting
    input Prediction graph P̂_G = (Ŝ, Ê)
    output Topologically sorted nodes T
    begin
        Vector T ← []
        Vector Worklist ← [ ŝ | ŝ ∈ Θ ]
        Set Visited ← Θ
        while Worklist ≠ [] do
            done ← true
            ŝ ← Top(Worklist)
            foreach (ŝ, t̂) ∈ Ê
                if t̂ ∉ Visited then
                    done ← false
                    Visited ← Visited ∪ {t̂}
                    Append(Worklist, t̂)
                end if
            end foreach

            if done = true then
                Prepend(T, ŝ)
                Pop(Worklist)
            end if
        end while
    end
end function
```

Given the topological sorting of the prediction graph and the fact that the prediction graph is a DAG, we can now compute a longest path through the graph. Note that the longest path through the prediction graph is not uniquely determined. There might be multiple paths through the graph with maximal but equal length.

For each node we remember the predecessor node and the maximal length of the path to the node that has been seen so far. Initially every node is associated with itself as predecessor and zero length. In topological order of the nodes we then update this information stepwise. If we find a longer path from the current node to one of its predecessors, we update the maximal path length and corresponding

predecessor node. After having visited all nodes, we construct a longest path.

First we choose the node with the maximum associated path length. To construct the longest path we follow the predecessors until we reach one of the initial hardware states. Algorithm 3.2 implements this computation in pseudo-code [22]. The algorithm returns a sequence of edges that describes a worst-case path through the prediction graph. For the prediction graph shown in Figure 3.5, the longest path algorithm computes the path $P = [\,(\hat{s}_0, \hat{s}_2), (\hat{s}_2, \hat{s}_5), (\hat{s}_5, \hat{s}_6), (\hat{s}_6, \hat{s}_7)\,]$.

**Algorithm 3.2** Computation of a Longest Path

```
function LongestPath
    input Prediction graph P̂_G = (Ŝ, Ê)
    output Longest path P
    begin
        Map Cost ← {ŝ → 0 | ŝ ∈ Ŝ}
        Map Predecessor ← {ŝ → ŝ | ŝ ∈ Ŝ}
        Vector T ← TopologicalSorting(P̂_G)
        foreach ŝ ∈ T
            c ← Cost(ŝ)
            foreach (ŝ, t̂) ∈ Ê
                c' ← Cost(t̂)
                if c + 1 > c' then
                    Cost ← Cost[ t̂ → c + 1 ]
                    Predecessor ← Predecessor[ t̂ → ŝ ]
                end if
            end foreach
        end foreach

        P ← []
        choose ŝ ∈ Ŝ where Cost(ŝ) ≥ Cost(t̂) for all t̂ ∈ Ŝ
        repeat
            t̂ = Predecessor(ŝ)
            Prepend(P, (t̂, ŝ))
            ŝ = t̂
        until ŝ ∈ θ
    end
end function
```

Reconsider the prediction graph shown in Figure 3.5. Starting with the abstract hardware state $\hat{s}_0$, the abstract state transition distinguishes between the successor states $\hat{s}_1$ and $\hat{s}_2$. For example, the transition $(\hat{s}_0, \hat{s}_1)$ could represent an initial cache miss, i.e., the local worst-case (LWC), whereas the state transition $(\hat{s}_0, \hat{s}_2)$ corresponds

to assuming an initial cache hit (i.e., non-LWC). Because the edge $(\hat{s}_0, \hat{s}_2)$ is part of the longest path $P$ and the opposing path $P' = [(\hat{s}_0, \hat{s}_1), (\hat{s}_1, \hat{s}_3), (\hat{s}_3, \hat{s}_4)]$ is shorter, the abstract simulation would reveal an instance of a timing anomaly.

To detect a timing anomaly we need to be able to identify whether an outcome of a split in the abstract state transition corresponds to the *local worst-case*. For example, if it is unknown whether a memory reference hits or misses the cache, we naturally identify the cache miss as the local worst-case. In many such situations we have an intuitive understanding about which decision is to be considered the local worst-case. For some others the local worst-case is not easily identified.

### 3.2.4 Non-Determinism

To derive safe timing guarantees a static timing analysis has to investigate the full (timing-relevant) behavior of the analyzed hardware architecture. In the presence of timing anomalies or domino effects, it is locally undecidable whether the assumption of a certain behavior will effectuate the global worst-case runtime performance. Hence, a static timing analysis investigates each and any possible behavior if it cannot decide which action would take place in the actual hardware.

Based on our investigation of embedded hardware architectures, we identify the following categories to classify the non-determinism (i.e., existence of a split) in the abstract state space:

**control-flow-induced** In general, embedded control-software is highly data dependent. The executed path depends on the current mode of operation, which is influenced by the input. For highly data-dependent programs, a static analysis is generally unable to restrict control-flow paths without further knowledge about the processed input. This kind of non-determinism is hardware-independent and thus not to be considered to cause timing anomalies.

**execution-induced** The analysis may not always argue about precise inputs of arithmetical computations. Hence, the analysis might not be able to determine whether an instruction finishes earlier due to an early-out optimization.

**memory-induced** We classify non-determinism as *memory-induced* if it is caused by memory accesses whose target address range is not precisely known. An unknown memory access could hit several memories with different timing behavior. Similarly, not knowing about the precise state of a memory can also provoke memory-induced non-determinism. For example, the access timing of an SDRAM access strongly depends on the previous access.

**cache-induced** Abstracting from buffer-like structures (e.g., caches), a static analysis is not always able to identify which data is actually stored in the concrete buffer. In this situation, the analysis has to consider both cases: a buffer hit and a buffer miss respectively.

A prominent example is the static analysis of caches that classifies memory references as whether they definitely miss the cache, surely hit the cache, or whether it is unknown if they hit or miss the cache [12]. Often, timing anomalies are observed in conjunction with caches.

**clock-induced** Many embedded hardware architectures feature several clock domains. For example, the processor core might by clocked at a different speed than the main memory. In the Freescale MPC5554 microprocessor, the external memory is driven at half or quarter the system clock. Asynchronous clock domains are also found in modern embedded processors with, e.g., a PCI controller that uses a dedicated PCI clock running at 33 MHz.

To simulate the timing behavior of an access to a differently clocked memory the clock jitter from the source (CPU) to the target (memory) domain needs to be known. If the distance between the rising edge of the clock signal in the target clock domain and the source clock domain is unknown, a static analysis has to assume every possibility. Especially for embedded systems with asynchronous clocks, this kind of non-determinism greatly increases the abstract state space.

| Category | Local Best-Case | Local Worst-Case |
|---|---|---|
| execution-induced | fastest execution | slowest execution |
| memory-induced | fastest access timing | slowest access timing |
| cache-induced | cache hit | cache miss |
| clock-induced | shortest clock distance | largest clock distance |

Table 3.2: *Classification of Non-Determinism:* The table lists the different classes of non-determinism that occur during abstract program simulation. For every class we can provide a basic intuition about which decision is considered the local best-case or the local worst-case respectively.

Table 3.2 summarizes these categories and discusses what would be considered to be the local best-case and the local worst-case respectively. Local best-case or local worst-case are not well defined for control-flow-induced non-determinism. Compared with the other categories, control-flow-induced non-determinism is a property of the analyzed program and is not related to a specific hardware feature. The major difference is that any decision regarding control-flow is related to the

program semantics. But, e.g., whether an arithmetical operation takes one or ten cycles to complete is not (directly) related to the semantics of the program.

Nonetheless, we do have to consider this class of non-determinism because the abstract program simulation has to take the corresponding hardware behavior into account. Depending on the hardware architecture, the pipeline analysis has to simulate branch prediction, or the effect of speculative execution. This naturally depends on the possible control-flow successors.

Usually we have a good intuition which state transitions are considered to be local worst-case. However, by means of action markers (see Definition 3.3 on page 33) we can formally identify which state transition is considered to be the local worst-case. We have to assume that for each state $\hat{s}$ with several successor states, i.e., $|\tau_{abs}(\hat{s})| > 1$ the outgoing edges are marked with action begin markers – otherwise there would have been no reason for the split to occur. Because $\gamma$ is action-deterministic (see Definition 3.8 on page 36: the abstract state automaton corresponds to a finite state automaton) each path through a transition that is marked with an action begin marker reaches a transition that is marked with the corresponding action end marker after a finite number of transitions. Thus, we can determine local worst-case transitions by means of a longest path search using Algorithm 3.2 on a prediction graph that is restricted to matching pairs of action begin and end markers. For this purpose we introduce the *$\lambda$-prediction graph*.

**Definition 3.13** ($\lambda$-Prediction Graph)
*Let $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton, $\hat{s} \in \hat{\mathcal{S}}$ with $|\tau_{abs}(\hat{s})| > 1$ and $\Lambda$ be a non-empty alphabet of actions. If there exists a transition $(\hat{s}, \hat{t}) \vdash \lambda$ for the action $\lambda \in \Lambda$ the* $\lambda$-prediction graph *for the state $\hat{s}$ is $\hat{\mathcal{P}}_{\hat{s}|\lambda} = (\hat{\mathcal{S}}, \hat{\mathcal{E}}_{\hat{s}|\lambda})$, where the set of edges is defined as follows:*

$$\hat{\mathcal{E}}_{\hat{s}|\lambda} := \bigcup_{\hat{t} \in \tau_{abs}(\hat{s}).(\hat{s},\hat{t}) \vdash \lambda} \tau_{\hat{s}|\lambda}(\hat{s}, \hat{t})$$

*where the he function $\tau_{\hat{s}|\lambda}(\hat{s}) : \hat{\mathcal{S}} \to 2^{\hat{\mathcal{S}} \times \hat{\mathcal{S}}}$ recursively follows state transitions until the expected action end marker is found.*

$$\tau_{\hat{s}|\lambda}(\hat{t}, \hat{u}) := \begin{cases} \{(\hat{t}, \hat{u})\} & \text{if } (\hat{t}, \hat{u}) \dashv_{\hat{s}} \lambda \\ \{(\hat{t}, \hat{u})\} \cup \bigcup_{\hat{v} \in \tau_{abs}(\hat{u})} \tau_{\hat{s}|\lambda}(\hat{u}, \hat{v}) & \text{otherwise} \end{cases}$$

We can now identify which decision in the abstract state space corresponds to the local worst-case (LWC). Figure 3.6 provides an example. The state transition $(\hat{s}_0, \hat{s}_1)$ is clearly the local wost-case transition with respect to the action $\lambda$, because it takes two transitions longer to reach the corresponding action end marker.

We formally define which transition is an *LWC transition* or a *non-LWC transition* respectively as follows.

**Definition 3.14** (LWC Transition)
*Let $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton, $\Lambda$ be a non-empty alphabet of actions, and $\hat{s} \in \hat{\mathcal{S}}$ with $|\tau_{abs}(\hat{s})| > 1$. A transition $(\hat{s}, \hat{t})$ is an* LWC transition *if for all actions $\lambda \in \Lambda$ with $(\hat{s}, \hat{t}) \vdash \lambda$ the transition is part of a longest path through the corresponding $\lambda$-prediction graph $\hat{\mathcal{P}}_{\hat{s}|\lambda}$. Otherwise, if there exists an action $\lambda \in \Lambda$ such that $(\hat{s}, \hat{t}) \vdash \lambda$ the state transition $(\hat{s}, \hat{t})$ is a* non-LWC transition.

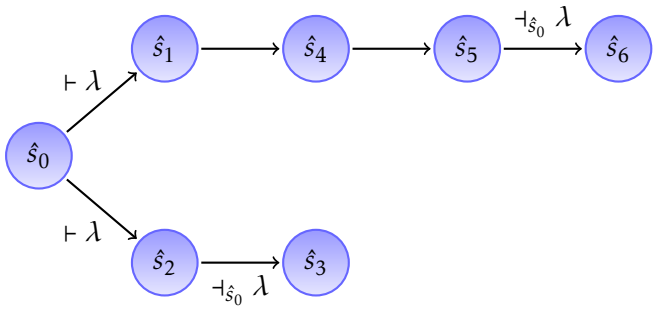



Figure 3.6: *$\lambda$-Prediction Graph:* $\lambda$-Prediction graph for the state $\hat{s}_0$. The state transition $(\hat{s}_0, \hat{s}_1)$ corresponds to the local worst-case because the path to reach the associated action end marker is longest.

### 3.2.5 Challenges for Static Analysis

The degree of non-determinism that is observed during abstract simulation strongly depends on the analyzed hardware. The complexity of a static timing analysis is directly related to the possible amount of non-determinism. Even if a hardware feature is perfectly well analyzable in isolation (e.g., a cache using the LRU replacement strategy) the combination of two such features might lead to a very costly or imprecise analysis. As a consequence, we observe an evolution of hardware states that would never occur during any real execution (i.e., analysis-induced non-determinism).

Take the Freescale MPC5554 hardware architecture as an example (see Section 6.1.5 on page 95). For the sake of simplicity we assume that the cache is updated using the LRU cache replacement policy.[7] LRU caches are perfectly well analyzable and proven to be free of cache-related timing anomalies or domino effects [4], i.e., a cache hit cannot effectuate a cache state where more subsequent cache misses can

[7] The actual hardware uses a pseudo-round robin replacement strategy with a global set counter.

occur for the same access sequence. Furthermore, the processor comprises an on-chip FLASH memory that offers a two-line read buffer to cache previously accessed FLASH pages. The size of a FLASH page corresponds to the size of a cache line. Both the FLASH memory read buffer and the LRU instruction cache are nicely analyzable in isolation. But a combination of both analyses will lead to an information loss due to the unfavorable structure of the combined analysis for the hierarchical layout of the MPC5554 hardware architecture.

A code fetch that targets the FLASH memory can either be cached or not. In case the requested instructions are already available in the instruction cache, the FLASH read buffer state will not be updated. Hence, we will lose any information about the read buffer state after a potential state join. After the state join the analysis is unable to tell whether an access to the same memory reference hits the FLASH read buffer. Figure 3.7 depicts this issue in detail.

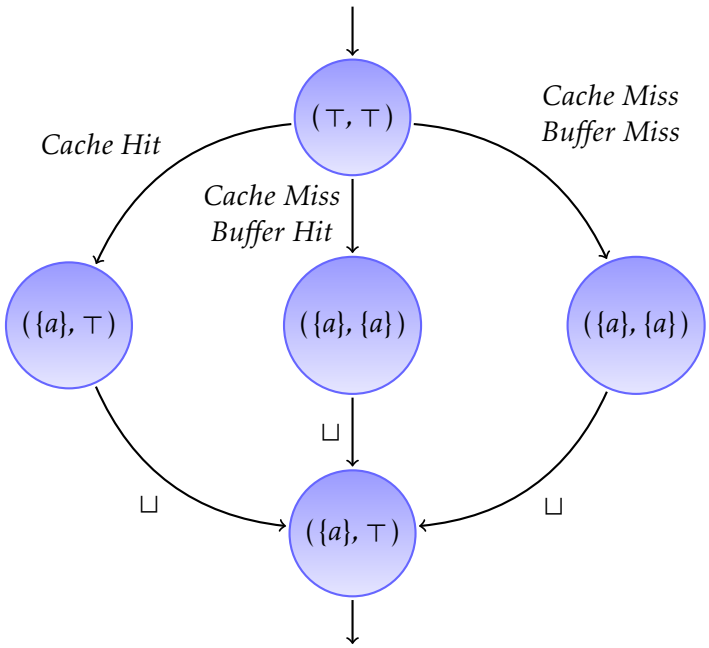



Figure 3.7: *Non-Determinism in Static Analysis:* Evolution of abstract pipeline state for an access to the internal FLASH memory. Each node shows a pair comprising an abstract cache state (left-hand side) and an abstract FLASH read buffer state (right-hand side). Initially we have no information about the state of both the cache and the FLASH read buffer, i.e., $(\top, \top)$. After the access is complete, the analysis knows that $a$ is definitely contained in the cache, but is unsure about the contents of the read buffer.

This problem is common to static analyses for architectures comprising components where an update of one system component does not affect the state of other involved

components under any assumption made. As long as the state of one such component is not precisely known any information about other components is inevitably lost at each join point. To avoid such loss of information, a static analysis would need to enumerate all possible states, which is usually infeasible in practice.

Consider the abstract state automaton $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ that provides an abstract model for a concrete processor. For each component of the real processor, i.e., functional units, caches, bus interconnects, etc., there exists a corresponding counterpart in the abstract model encoded by $\hat{\mathcal{A}}$. Hence, the abstract state space can be understood as a composition of abstract component states, i.e., $\hat{\mathcal{S}} = \hat{\mathcal{S}}_{\mathcal{C}_1} \times \cdots \times \hat{\mathcal{S}}_{\mathcal{C}_n}$, where $\hat{\mathcal{S}}_{\mathcal{C}_i}$ is the abstract state space for the component $\mathcal{C}_i$ where $1 \leq i \leq n$ and $n \in \mathbb{N}$ is the number of processor components.

An abstract state transition does not necessarily update every abstract component state. Which component states are affected strongly depends on the interaction between individual hardware components [42].

To discover the above described problem in the abstract state space we first need to identify the modified components. For this purpose we define the *state transition difference*.

**Definition 3.15** (State Transition Difference)
*Let $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton, where $\hat{\mathcal{S}}$ is the composition of abstract component states, i.e., $\hat{\mathcal{S}} = \hat{\mathcal{S}}_{\mathcal{C}_1} \times \cdots \times \hat{\mathcal{S}}_{\mathcal{C}_n}$ with $n \in \mathbb{N}$. The* state transition difference *$\tau_{diff} : \hat{\mathcal{S}} \times \hat{\mathcal{S}} \to \mathbb{B}^n$ is defined for $\hat{s} = (\hat{c}_1, \ldots, \hat{c}_n)$ and $\hat{t} = (\hat{c}'_1, \ldots, \hat{c}'_n) \in \tau_{abs}(\hat{s})$ as follows:*

$$\tau_{diff}(\hat{s}, \hat{t}) := (b_1, \ldots, b_n) \text{ with } b_i := \begin{cases} 1 & \text{if } \hat{c}_i \neq \hat{c}'_i \\ 0 & \text{otherwise} \end{cases}$$

By means of the state transition difference we can observe which abstract components a state transition are affected. If for a given abstract state $\hat{s} \in \hat{\mathcal{S}}$ all state transitions $(\hat{s}, \hat{t})$ with $\hat{t} \in \tau_{abs}(\hat{s})$ affect the same hardware components the state transitions originating at $\hat{s}$ are *inclusive*. Otherwise the state transitions originating at $\hat{s}$ are *non-inclusive*.

**Definition 3.16** (Inclusive State Transition)
*Let $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton, where $\hat{\mathcal{S}}$ is the composition of abstract component states. Let $\hat{s} \in \hat{\mathcal{S}}$ be an abstract state. The state transitions originating at $\hat{s}$ are* inclusive *iff for all $\hat{t}$ and $\hat{u} \in \tau_{abs}(\hat{s})$ it holds $\tau_{diff}(\hat{s}, \hat{t}) = \tau_{diff}(\hat{s}, \hat{u})$. Otherwise the state transitions originating at $\hat{s}$ are* non-inclusive.

Non-inclusive state transitions imply information loss in the abstract state space due to later abstract state joins. Reconsider the example in Figure 3.7. Here the state transitions originating at the initial hardware state are non-inclusive. Due to

the state join at the final hardware state the analysis loses all information about the second hardware component (i.e., the FLASH read buffer).

This phenomenon is not uncommon to static timing analysis – especially for the analysis of complex hardware architectures. Non-inclusive state transitions cannot always be avoided, either due to limitations of the analysis tool chain or simply because of the hardware architecture. The following list comprises common cases of non-inclusive state transitions:

**value analysis precision** The precision of the value analysis has a major impact on the performance of a static timing analysis. For instance, if the target of a memory access cannot be precisely identified, the analysis has to consider all possible memories as a potential target. If updating the memory state has an impact on the timing behavior of subsequent memory accesses non-inclusive state transitions cannot be avoided. The processor to memory bus clock jitter, read buffers (see above), are just some examples.

**virtual memory** A related problem occurs in the static analysis of virtual memory. Typically a translation lookaside buffer (TLB) is employed to cache the virtual to physical address translation attributes (i.e., page table entries). Upon a TLB miss an interrupt is triggered and the corresponding interrupt service routine (re)loads the requested page table entry. If the state of TLB is initially unknown the state of other (involved) hardware components remains (partially) unknown until the TLB contents are precisely known. Naturally, this greatly impairs the precision and resource consumption of a static timing analysis.

**non-inclusive caches** This phenomenon has been already discussed in Figure 3.7. Cache-related non-inclusive state transitions are unavoidable if the assumption of a cache hit does not affect the same system components that are updated upon a cache miss otherwise.

For example, consider a Freescale MPC755 processor that is connected to a DRAM controller. As long as the state of the processor's L1 cache is unknown, a static analysis cannot gather any information about the status of the DRAM controller and is thus unable to predict any DRAM page hits.

This phenomenon cannot be avoided completely because either the value analysis precision cannot be further improved, or a modification of the hardware architecture is not possible at all, or, if possible, would incur an unacceptable degradation of performance. However, some hardware architectures, like the Freescale MPC7448, features inclusive L1 and L2 caches. Any modification done to the L1 cache automatically implies a state update of the (outer) L2 cache. For such architectures, the static analysis will not lose information about the L2 cache state after state joins due to an unknown state of the L1 cache.

# Timing Anomalies and Domino Effects 4

Intuitively, a timing anomaly is a counterintuitive behavior of a hardware architecture where a local speed-up leads to a global slow-down. Various – average-performance increasing – hardware features may exhibit this kind of non-local execution time behavior. Caches or similar buffer-like hardware components are often involved.

For example, a division operation may take between six and eleven processor cycles to complete depending on the input data. Depending on the available hardware features, the overall execution might be slower if the instruction completes earlier. Here, the timing anomaly is caused by not having any knowledge about the divisor or the numerator respectively. We classify the cause for this anomaly as *execution-induced* according to Section 3.2.4 on page 43.

In the following we formally define the term timing anomaly in accordance with the definition found in [34]. By means of the prediction graph as defined in Definition 3.12 on page 40 we are able to detect instances of timing anomalies that occur during the abstract simulation of a program. Furthermore we define the notion of a domino effect and stress its difference from bounded timing anomalies.

## 4.1 Formal Definition

By means of the definition of non-LWC transitions (see Definition 3.14 on page 46) we can formally define the term timing anomaly as follows.

**Definition 4.1** (Timing Anomaly)
*Let* $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ *be an abstract state automaton. The abstract hardware model exhibits a* timing anomaly *if there exists a program represented by the control-flow graph* $G$ *such that a path* $\omega$ *through the prediction graph* $\hat{\mathcal{P}}_G$ *contains a non-LWC state transition* $(\hat{s}, \hat{t})$ *whereas all other paths* $\omega'$ *through* $\hat{s}$ *are shorter, i.e.,* $|\omega| > |\omega'|$*.The state* $\hat{s}$ *is called* timing-anomalous.

Using the observations made in the abstract state space, we can infer necessary preconditions to observe the timing-anomalous behavior on the concrete hardware. If it is possible to effectuate initial hardware states such that the expected timing anomaly is measurable in concrete executions of the program, we have detected a real instance of a timing anomaly. Otherwise, the chosen abstraction has lost necessary information to avoid this anomalous hardware behavior. Such timing anomalies are *virtual timing anomalies*. The loss of such information is in general unavoidable, as discussed in Chapter 2.

**Algorithm 4.1** Detection of Timing Anomalies

**function** DetectTimingAnomalies
**input** Prediction graph $\hat{\mathcal{P}}_G = (\hat{\mathcal{S}}, \hat{\mathcal{E}})$
**output** Timing-anomalous states $S$
**begin**
$\omega' \leftarrow \text{LongestPath}(\hat{\mathcal{P}}_G)$
**repeat**
$\omega \leftarrow \omega'$
$S \leftarrow \{\hat{s} \mid \exists \hat{t} \in \hat{\mathcal{S}} . (\hat{s}, \hat{t}) \in \omega \wedge (\hat{s}, \hat{t}) \text{is non-LWC}\}$
$\hat{\mathcal{P}}_G \leftarrow (\hat{\mathcal{S}}, \hat{\mathcal{E}} \setminus \{(\hat{s}, \hat{t}) \in \hat{\mathcal{E}} \mid \hat{s} \in S\})$
$\omega' \leftarrow \text{LongestPath}(\hat{\mathcal{P}}_G)$
**until** $S = \emptyset \vee |\omega| \neq |\omega'|$
**end**
**end function**

Based on the definitions and by means of the longest path algorithm (see Algorithm 3.2) we can then implement an iterative search to detect instances of timing anomalies during the abstract program simulation. Given a prediction graph, we first compute a longest path $\omega$ through the prediction graph. Afterwards we eliminate all non-LWC transitions from the prediction graph that are present on that longest path. If there are no such transitions on the computed path, we have not found a timing anomaly instance. Otherwise, we compute a longest path $\omega'$ through the restricted prediction graph. If $\omega'$ has the same length as $\omega$, we repeat the above steps. If not, $\omega$ describes a worst-case execution of a program that exhibits a timing anomaly in the abstract state space. If the set of timing-anomalous states is non-empty, we have found an instance of a timing anomaly in accordance with Definition 4.1. Algorithm 4.1 implements this algorithm in pseudo-code.

In this fashion we are able to automatically detect instances of timing anomalies for programs with a finite number of paths. Section 6.2 on page 108 presents our findings for seven different hardware architectures.

## 4.2 Infinite Programs

So far, we have restricted ourselves to finite programs, i.e., programs that only comprise paths of finite length through the control-flow graph. A timing anomaly in accordance with Definition 4.1 can thus only have a bounded impact on the overall timing behavior. In general however, programs may feature infinite paths (e.g., unbounded loops, or similar). For such programs we can no longer argue about the length of the longest path through the corresponding prediction graph. Thus we

need to reason about paths of finite length $n$ through the CFG to cope with arbitrary programs.

**Definition 4.2** ($n$-Prediction Graph)
*Let $G = (V, v_s, E)$ be a CFG, $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ an abstract state automaton, and $\theta \in 2^{\hat{\mathcal{S}}}$ be the set of initial abstract states.*

*An $n$-*prediction graph *is a prediction graph that is restricted to paths through the control-flow graph of length $n \in \mathbb{N}$. Symbolically: $\hat{\mathcal{P}}_{G|n} = (\hat{\mathcal{S}}, \hat{\mathcal{E}}|n)$, where $\hat{\mathcal{E}}|n = \{(\hat{s}, \hat{t}) \mid \hat{s} \rightsquigarrow_{\pi} \hat{t} \wedge |\pi| = n\}$.*

Often the impact of a timing anomaly on the execution behavior stabilizes eventually. This means that the difference between an execution that comprises a non-LWC decision and any other execution is bounded by a constant. Such a timing anomaly is called *$k$-bounded timing anomaly*, where $k$ is the maximal difference caused by the timing anomaly. This is formalized in Definition 4.3.

In the presence of a $k$-bounded timing anomaly, a static timing analysis could always assume the local worst-case, adding the constant $k$ to the computed WCET bound [33]. The timing-anomalous hardware state determines the actual impact on timing, which is not easily computable. In most cases the precision of a static timing analysis will degrade by assuming the local worst-case and adding the constant $k$ to the computed WCET bound.

**Definition 4.3** ($k$-bounded Timing Anomaly)
*Let $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton. The abstract hardware model exhibits a $k$-*bounded timing anomaly *if there exists a program $G$, a non-LWC transition $(\hat{s}, \hat{t})$, and a constant $k \in \mathbb{N}$ such that any $n$-prediction graph $\hat{\mathcal{P}}_{G|n}$ that contains a path $\omega$ through $(\hat{s}, \hat{t})$ only comprises paths $\omega'$ through $\hat{s}$ that are at most $k$ shorter, i.e., $|\omega| > |\omega'| \wedge |\omega| - k \leq |\omega'|$. The abstract state $\hat{s}$ is $k$-*timing-anomalous.

Unfortunately, some hardware features cause timing anomalies whose effects on timing are unbounded. Such timing anomalies are known as *domino effects*. Domino effects are essentially different from $k$-bounded timing anomalies: A $k$-bounded timing anomaly occurring in a loop only has a limited timing effect that eventually stabilizes. In other words, the loop body runtime will only differ for a bounded number of iterations and converge finally. In the presence of a domino effect, the loop body runtime will take different values without convergence in the future.

Contrary to $k$-bounded timing anomalies, a non-LWC transition is not a necessary precondition for a domino effect to occur. Furthermore we need to fix the executed path through $G$ after a specific instruction view $v$. This is necessary to avoid the control-flow to "cause" a domino effect. Consider an `if-then-else` statement where the `then` part enters an unbounded loop and the `else` part exits the program. If

the condition is statically unknown both possibilities have to be considered. It is obvious that the difference between the `then` and the `else` part cannot be bounded by any $k \in \mathbb{N}$. Definition 4.4 provides a formal definition.

**Definition 4.4** (Domino Effect)
*Let $\hat{\mathcal{A}} = (\hat{\mathcal{S}}, \tau_{abs})$ be an abstract state automaton. Let $G = (V, v_s, E)$ be a control-flow graph, $\pi_{post}$ an arbitrary sequence of instructions and $v \in V$ an arbitrary instruction and $\Pi = \{\pi \mid \pi = \pi_{pre} \circ v \circ \pi_{post}$ is a path through $G\}$. The set $\Pi$ is a collection of paths that share the tail $\pi_{post}$ following $v$. The* $n$-prediction graph restricted to $\Pi$ *is then defined as: $\hat{\mathcal{P}}^{\Pi}_{G|n} = (\hat{\mathcal{S}}, \hat{\mathcal{E}}|^{\Pi}_{n})$, where $\hat{\mathcal{E}}|^{\Pi}_{n} = \{(\hat{s}, \hat{t}) \mid \hat{s} \rightsquigarrow_{\pi} \hat{t} \wedge |\pi| = n \wedge \exists \pi' \text{ s.t. } \pi \circ \pi' \in \Pi\}$.*

*The abstract hardware model exhibits a* domino effect *if for any $k \in \mathbb{N}$ there is an $n \in \mathbb{N}$ such that the difference between a longest path $\omega$ through $\hat{\mathcal{P}}^{\Pi}_{G|n}$ and any other path $\omega'$ cannot be bounded by $k$, i.e., $|\omega| - k > |\omega'|$.*

Domino effects are real. Schneider [38] has demonstrated that the MPC755 pipeline actually causes a domino effect. Furthermore, Berg [3] was able to show that, in contrast to the LRU replacement policy, the pseudo-LRU, the FIFO, and the round-robin replacement strategies suffer from domino effects. In Section 4.5 we discuss a cache replacement policy where a non-LWC transition is not a necessary precondition for a domino timing anomaly to occur. In Section 6.2.4 on page 112 we demonstrate a domino effect that is present in the MPC565 processor.

The presence of timing anomalies impacts both performance and precision of a static timing analysis. In general, an analysis cannot always choose the most expensive execution, as this decision might not always lead to the global worst-case execution time. Consequently, the number of states to consider during analysis time might increase greatly if the absence of timing anomalies cannot be proven for an analysis state where multiple successor states are possible.

The inability of proving the absence of timing anomalies might also lead to an increase in the computed WCET bound. Section 4.5 discusses an anomaly present in the LEON2 processor that can lead to an overestimation of up to 20%, which strongly depends on the code structure of the analyzed program.

## 4.3 Classification of Timing Anomalies

The classification of Reineke et al. [34] categorizes timing anomalies according to the hardware property that is responsible for the timing anomaly. Reineke discerns three different classes of timing anomalies:

**scheduling timing anomaly** Most timing anomalies found in the literature belong to this class. Depending on the execution time of a task, a faster execution

might lead to a globally longer schedule. Such anomalies are well-known in the scheduling domain and have been thoroughly studied on various scheduling routines.

**speculation timing anomaly** Some timing anomalies are caused by speculative behavior, i.e., mechanisms that attempt to predict future behavior, such as the direction of control-flow. For example, a prefetching processor in combination with an instruction cache may cause such a timing anomaly. After an initial cache hit the processor might speculatively prefetch into the wrong direction and thus delay the overall execution due to a cache miss. A speculation anomaly is found in the LEON2 processor.

**cache timing anomaly** Cache timing anomalies are caused by some non-LRU cache replacement strategies. We further discern *bounded cache timing anomalies* and *cache domino effects*. Several cache replacement algorithms have been proven to cause domino effects, such as FIFO or P-LRU [3].

The timing anomaly class by itself does not suffice to completely understand the anomaly's nature. Often an additional hardware feature is responsible for the timing anomaly to occur. Thus we also need to consider the non-determinism present in the abstract state space (see Section 3.2.4 on page 43). For example, a speculation timing anomaly is mostly caused by an initial cache hit that then triggers an additional code fetch that misses the cache. In this case, the timing anomaly is cache-induced. However, it is also possible that an instruction with variable execution time causes this behavior. If that is the case, we call the timing anomaly an execution-induced speculation timing anomaly.

To understand the impact on static timing analysis, we need to know the kind of timing anomaly, i.e., whether it is a $k$-bounded or an unbounded timing anomaly. As hinted earlier, we can implement a static timing analysis for a hardware architecture that has a $k$-bounded timing anomaly without considering every possibly transition if the constant $k$ is known. Instead we could always focus on LWC transitions and add the constant $k$ to the computed WCET bound per instance of the $k$-bounded timing anomaly. On the other hand, a static analysis for a processor that suffers from a domino effect always has to consider every possible transition in the abstract state space. Otherwise, the estimated WCET bounds cannot be guaranteed to be safe.

Arguing that a timing anomaly is $k$-bounded is challenging. By means of the $n$-prediction graph we can only collect evidence for paths of finite length through a specific program. Computing a constant $k$ that holds for every program is hardly possible. Furthermore, there might be some program that shows a domino effect under very similar conditions. Besides, a hardware feature that can trigger a domino effect might sometimes only cause a constantly-bounded timing anomaly depending on the execution history.

The above definitions allow for a better understanding of this phenomenon and its influence on static WCET analysis. The different types of timing anomalies allow for a classification of hardware architectures.

## 4.4 Classification of Architectures

Depending on whether a hardware architecture suffers from $k$-bounded, unbounded or no timing anomalies, the architecture belongs to one of the following classes. We adopt the categories proposed by Wilhelm et al. [47]:

**fully timing compositional architecture** The architecture does not exhibit any timing anomaly. Hence, a static timing analysis can safely follow local worst-case paths only. An example for this class of architectures is the ARM7 processor. On a timing accident all components of the pipeline are stalled until the accident is resolved. This even allows for a much simpler analysis where architecture components (e.g., cache, bus occupancy, etc.) can be analyzed separately, i.e., a safe parallel decomposition of the WCET problem is feasible.

**compositional architecture with $k$-bounded effects** The architecture exhibits a $k$-bounded timing anomaly but does not suffer from domino effects. In general, a WCET analysis would have to consider all decisions to compute safe execution time bounds. To trade precision with efficiency, it would be possible to safely discard non-LWC paths by adding a constant $k$ per (possible) occurrence of a timing anomaly to the computed WCET bound. So far, no hardware architecture has been formally proven to belong to this class.

**non-compositional architecture** A hardware architecture that belongs to this class exhibits domino effects. The MPC755 is known to belong to this class of architectures because its complex pipeline might cause a domino effect. For such architectures timing analyses always have to follow all paths since any local effect may influence the future execution arbitrarily.

## 4.5 Examples

Figure 4.1 gives an example of a cache-induced speculation timing anomaly caused by the interaction between the branch prediction mechanism, the instruction cache, and the processor's ability to execute instructions out-of-order. In this example the processor is currently executing a conditional change-of-flow instruction whose condition is not yet evaluated at the moment the instruction $A$ is about to be fetched. Upon a cache hit for code fetch of instruction $A$, the processor starts to speculatively fetch the uncached branch target $B$. Although the initial cache hit locally causes a

faster execution, the overall execution is slowed down, because the cache line fill fetching $B$ takes longer than resolving the branch condition.

This timing anomaly could also be caused by speculative execution. This means that the processor starts to execute the fetched instructions, while the processor computes the branch condition. Instead of fetching the instruction $B$ and being stalled due to a cache miss, the processor could speculatively execute the previously fetched instruction $B$ resulting in a longer stall of the processor's pipeline.

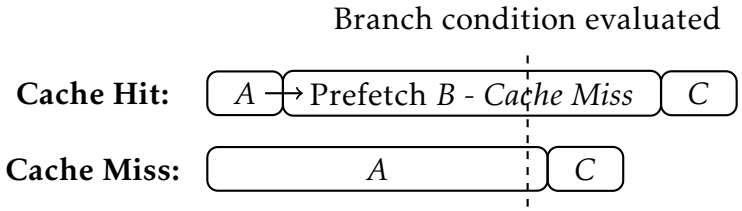



Figure 4.1: *Cache-Induced Speculation Timing Anomaly:* The processor executes a conditional branch instruction whose condition is yet unresolved. Assuming a cache hit for the initial code fetch, the processor speculatively fetches the instruction $B$ that is not contained in the cache. This causes an overall longer execution time because the cache line fill operation stalls the processor longer than it takes to resolve the branch condition.

Figure 4.2 demonstrates an execution-induced scheduling timing anomaly (e.g., caused by dividers with an early-out mechanism). Here, the processor features two execution units, an arithmetical logical unit (ALU) and a floating point unit (FPU). Depending on the input parameters, the ALU executes integer division instructions, like $a_1$, quicker. Completing instruction $a_1$ earlier, the processor is able to dispatch instruction $f_2$ before instruction $f_1$. This effectively causes the processor to execute all instructions sequentially. The instruction sequence takes longer to complete, because the processor cannot benefit from its ability to execute instructions in parallel. On the contrary, if instruction $a_1$ takes longer to complete, the processor will dispatch instruction $f_1$ earlier. This allows the processor to execute the instructions $a_2$ and $f_2$ in parallel, resulting in an overall faster execution.

The variable-execution-time-triggered timing anomaly corresponds to a so-called scheduling anomaly. In the same fashion, a task that terminates earlier could lead to an overall longer schedule, whereas a faster schedule could be achieved if the very same task would run to completion a bit later. Greedy schedulers are usually unable to prevent this kind of anomaly.

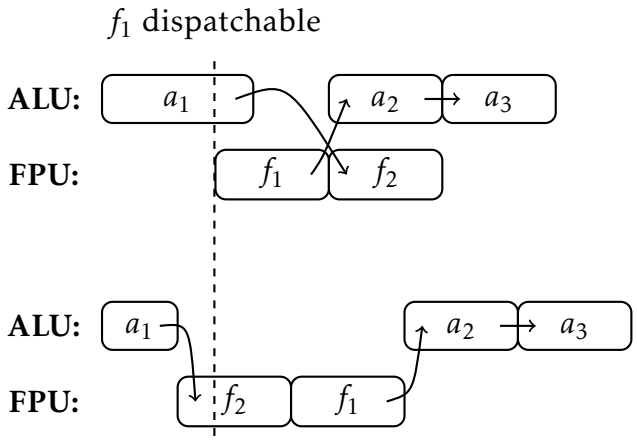



Figure 4.2: *Execution-Induced Scheduling Timing Anomaly:* This example demonstrates that a fast instruction execution might cause a global slow-down of the whole instruction sequence. The instructions $a_1$, $a_2$, and $a_3$ execute on the ALU. The ALU features an early-out mechanism that allows integer divide instructions, such as $a_1$, to complete faster under certain circumstances. The other instructions $f_1$ and $f_2$ solely execute on the FPU. Edges between instructions indicate definition-use dependencies.

### LEON2 Timing Anomaly

In this section we discuss the LEON2 hardware architecture. The LEON2 was developed at Aeroflex Gaisler as a successor of the ERC32 processor. A radiation-hardened version of the LEON2 is available [1] which makes it suitable for the space domain.

The LEON2 features a rather simple pipeline that comprises five stages. To speed up execution the LEON2 comprises disjoint instruction and data caches. Figure 4.3 depicts a block diagram of the LEON2 showing the memory hierarchy.

On a first view, the LEON2 appears to be a fully timing compositional architecture. The processor neither performs speculative fetching nor does it execute instructions speculatively. The LEON2 does not possess any branch-prediction mechanism. Instructions are executed and completed in-order. Each instruction has to visit the five pipeline stages one after another. Thus, an instruction cannot overtake a slower instruction blocking a certain pipeline stage. This prevents the possibility of a scheduling anomaly. Upon a timing accident (i.e., a cache miss) the internal pipeline is stalled until the accident is resolved. Both caches commonly use the LRU replacement policy[8], which is known to behave in a timing compositional manner.

[8]The LEON2 is synthesized from a VHDL model where different replacements algorithms can be configured.

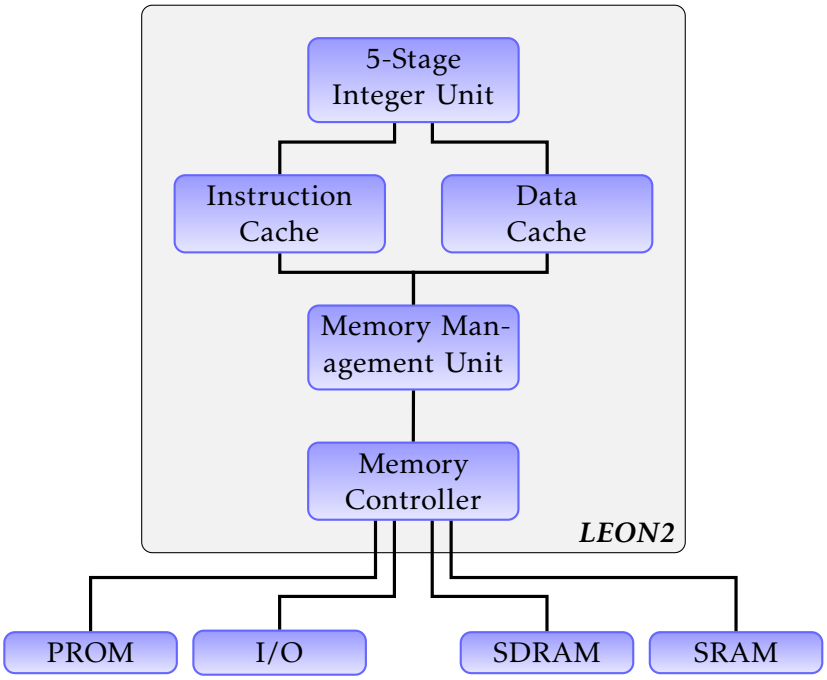



Figure 4.3: *Simplified Block Diagram of the LEON2 Architecture:* The LEON2 core comprises a five stage integer unit that is connected to an instruction cache and a data cache. The memory management unit (if present) implements memory protection. Communication with I/O devices and memory modules is established by the memory controller, which supports FLASH, SRAM and SDRAM memories.

It appears that none of the above described timing anomalies can occur. However, in the following we will show that the LEON2 has a hardware feature that potentially triggers a timing anomaly (depending on the system state).

Upon a cache miss, the processor needs to load the missing cache line from main memory. Usually, the whole cache line is loaded and put into the cache. Until the cache line has been filled, the processor stalls the originating memory access. To reduce latencies, some architectures start loading the cache line at the requested address directly forwarding the received data to the core (*cache streaming*).

A similar technique is available in the LEON2 architecture. Each cache line is equipped with valid bits for each word[9] inside the cache line. A cache line is either 16 or 32 byte wide and thus comprises either four or eight valid bits. Upon a data cache miss, solely the requested word is loaded from memory and put into the corresponding cache line. The instruction cache operates slightly differently from

[9] A word is four bytes on the LEON2 hardware architecture.

the data cache. If an instruction fetch misses the code cache, the processor burst-fills the corresponding cache line starting from the requested instruction till the end of the line. The processor does not issue wrap-around burst fetches. Consequently, cache lines might only be filled partially. Furthermore, the processor does not check for existing entries upon burst-filling the cache line. A timing anomaly finally becomes possible, as the LEON2 processor allows cache line fills to be interrupted under certain circumstances [20].

Figure 4.4 demonstrates how the cache line fill mechanism can trigger a timing anomaly. In this example the contents of the cache are assumed to be initially unknown. Each cache line can hold up to eight instructions. Assuming an initial cache miss, the core fills the whole cache line. All in all, the processor issues eight instruction fetches. Assuming cache hits for the first two instruction fetches (basic block $A$) causes a timing anomaly. The remainder of the target cache line still remains unknown. Reaching the basic block $C$, a static analysis then would need to assume a cache miss. Recall that the processor might abort a cache line fill operation. Thus, the instructions of basic block $C$ need not necessarily be cached, although cache hits have been assumed for the initial accesses to the cache line. In this case, the core will fill the upper half of the target cache line. Eventually, the program branches to the basic block $B$. Again, a static analysis would need to assume a cache miss. Because the processor does not check whether burst-fetched instructions are already cached, the instructions in basic block $C$ will be fetched again. Altogether the core performs ten fetches after the initial cache hits. So, the processor performs 25% more memory accesses under the initial assumption of a hit.

Despite the simple structure of the LEON2 a timing anomaly is possible, caused by a rather simple, average-case performance increasing hardware feature. Obviously, the timing anomaly is a speculation timing anomaly (see Section 4.3). Fetching subsequent instructions upon an instruction cache miss, the processor assumes a sequential execution of the program.

The timing anomaly is $k$-bounded. It is easy to see that the described effect will eventually stabilize – a positive side effect of the LRU cache replacement policy.

#### SIMPLE-MRU Domino Effect

This section discusses an artificial replacement policy SIMPLE-MRU by means of which we demonstrate that a non-LWC transition is not a necessary precondition for a domino effect to occur. Contrary to the LRU replacement strategy, SIMPLE-MRU discards the most-recently accessed cache line upon a cache miss.

Figure 4.5 demonstrates the domino effect by means of a 2-way cache using the SIMPLE-MRU replacement strategy. In this example, the memory locations $a$ and $b$ are accessed in an alternating pattern. Starting with a cache that already contains $a$,

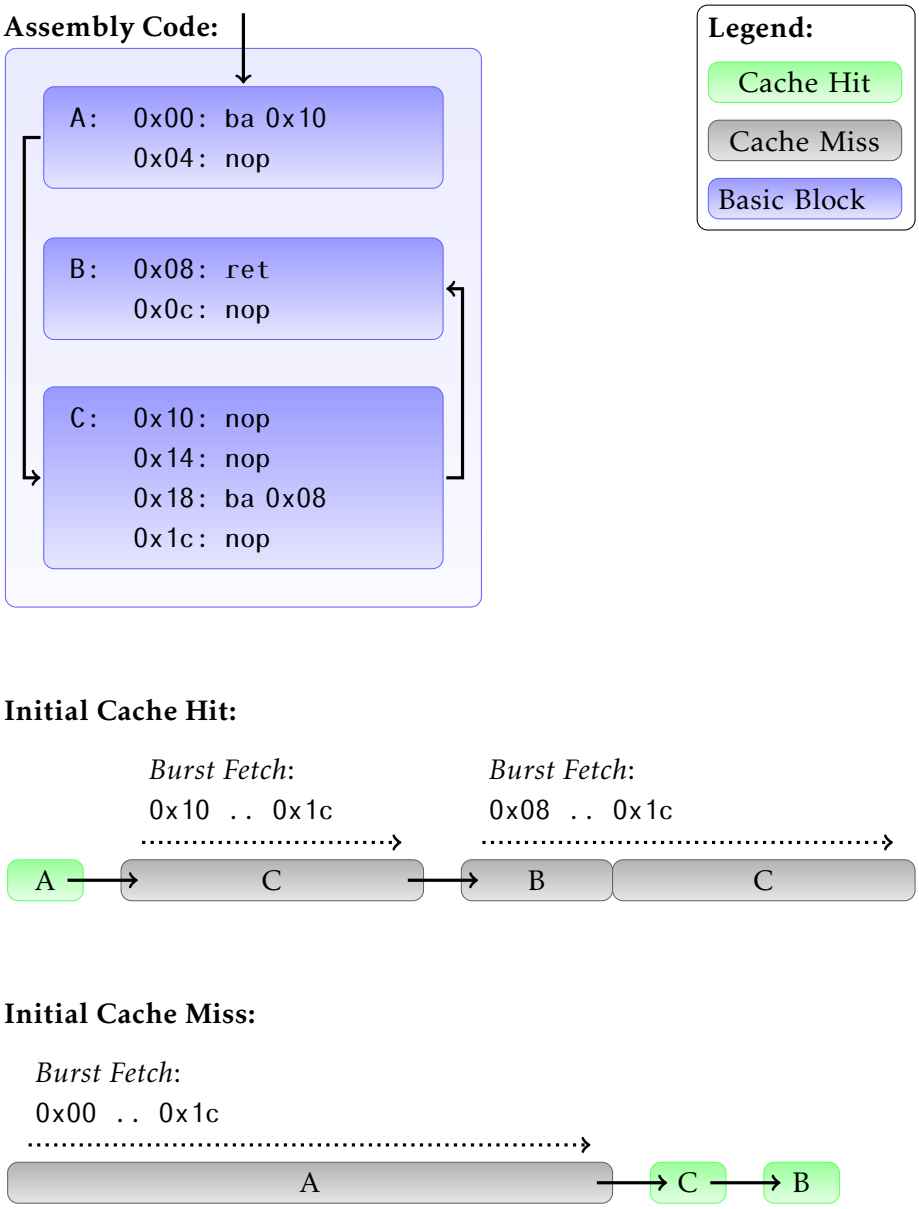



Figure 4.4: *LEON2 Timing Anomaly:* The example demonstrates a timing anomaly present in the LEON2 processor caused by the instruction cache line fill mechanism. The basic blocks A, B, and C reside in the same cache line. The local best-case – assuming a cache hit for the instructions in basic block A – causes the global worst-case execution of the example: The core performs ten instructions fetches. On the contrary, only eight instruction fetches are issued upon an initial cache miss.

the cache set contents stabilize after two accesses. After the first two cache misses the repeating access sequence will only produce cache hits. Starting with a cache set that contains the addresses $a$ and $c$, where $a$ is the most-recently accessed one, each access to the cache except for the first will lead to a cache miss. Because SIMPLE-MRU retains older data (i.e., the memory location $c$ in this case), an access to $a$ will evict $b$ from the cache and vice versa.

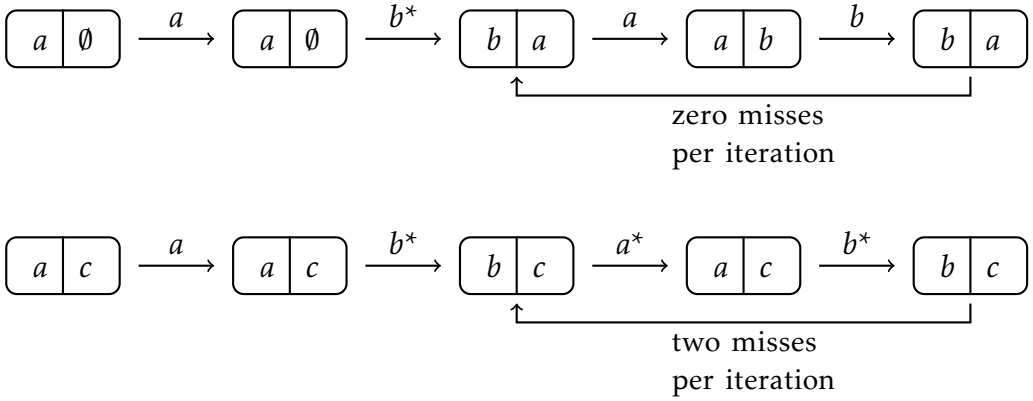



Figure 4.5: *Domino Effect for SIMPLE-MRU Replacement Policy:* 2-way cache using a SIMPLE-MRU replacement policy for the repeating access sequence $(a, b)^+$. The left-hand side of a set depicts the most-recently accessed element. The first row features a partially filled cache, where no misses occur for the given sequence except for the first access. The second row demonstrates a different initial cache state that causes all accesses except the first to miss the cache. Each miss is marked by $^*$.

Whether a static WCET analysis computes safe and precise bounds depends on the abstract hardware model. In the best-case the abstract model is obtained from a behavioral processor description, such as a VHDL model [37]. For hardware architectures where no such model is available, the analysis designer has to rely on processor manuals and measurements to verify the correctness of the abstract model. Here, we propose means to automatically compare static analyses with measurements.

## 5.1 Methodology

We extend the prediction graph (see Definition 3.12) such that it allows for an automatic comparison between the prediction and the measured execution behavior. We annotate the edges in the prediction graph with events that can be measured on the real device while executing the program. The granularity at which the comparison takes place depends on the debug facilities provided by the hardware. Section 5.2 discusses the different levels of granularity.

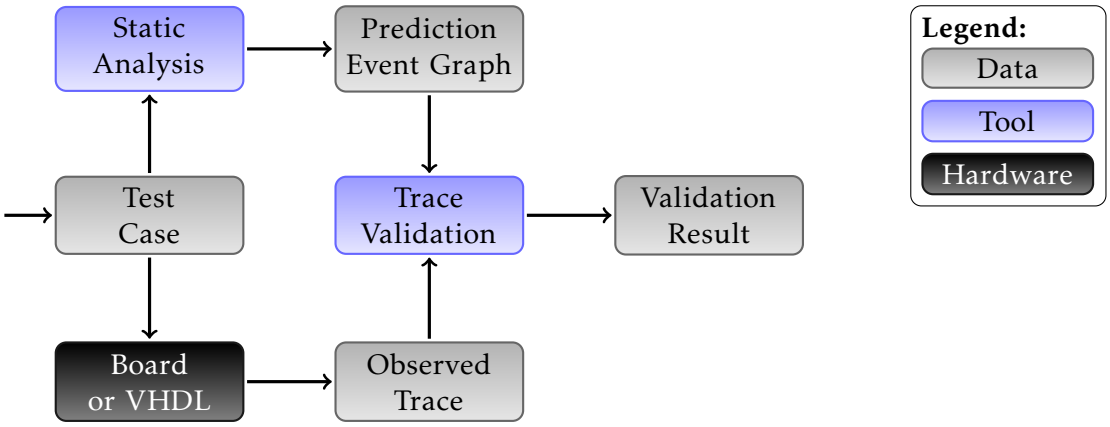



Figure 5.1: *Trace Validation Procedure:* A test case is the starting point of the trace validation. First we obtain the predicted execution behavior by means of static analysis. Second we measure the execution behavior on the real device or simulate via an VHDL model. Finally we compare prediction and measurement.

Figure 5.1 provides a top-level view on the comparison methodology. For a test case we perform both static WCET analysis and measurement to obtain base data for trace validation. A graph search then determines whether the measured trace

of events is contained in the prediction graph. The trace validation is successful if there exists a path through the prediction graph that comprises the events in exactly the same order in which they have been observed. To describe the comparison between measurement and prediction we require a formalism to compare concrete (i.e., measured) and abstract (i.e., predicted) events.

We choose the set of concrete events $\Sigma$ according to the measurement capabilities of the analyzed hardware. Events that cannot be observed (e.g., state changes of internal processor signals) are irrelevant for trace validation. Section 5.2 discusses this in more detail.

**Definition 5.1** (Concrete Trace)
*Let $\Sigma$ be a non-empty set of concrete events. A* concrete trace $\rho \in \Sigma^+$ *is a non-empty sequence of concrete events. The* empty event $\perp \in \Sigma$ *signifies that no event has occurred.*

Similar to program simulation we introduce the set of abstract events $\hat{\Sigma}$ with the partial order $\sqsubseteq$. For soundness reasons we require a monotone *event concretization function* $\gamma : \hat{\Sigma} \to 2^{\Sigma}$ that maps abstract events to sets of concrete events. The monotonicity of $\gamma$ ensures that the partial order $\sqsubseteq$ on $\hat{\Sigma}$ arranges events according to their precision.

**Definition 5.2** (Abstract Event Match)
*Let $\sigma \in \Sigma$ be a concrete event, $\hat{\sigma} \in \hat{\Sigma}$ be an abstract event, and $\gamma : \hat{\Sigma} \to 2^{\Sigma}$ be an event concretization function. The abstract event $\hat{\sigma}$* matches *the concrete event $\sigma$, written as $\hat{\sigma} \succ \sigma$, iff $\sigma \in \gamma(\hat{\sigma})$. It holds $\hat{\sigma} \succ \perp$, iff $\gamma(\hat{\sigma}) = \emptyset$, i.e., no abstract event has occurred.*

**Definition 5.3** (Predicted Abstract Trace)
*Let $\hat{\Sigma}$ be a set of abstract events, $\hat{\mathcal{P}}_G = (\hat{\mathcal{S}}, \hat{\mathcal{E}})$ be a prediction graph, and $\delta : \hat{\mathcal{E}} \to \hat{\Sigma}$ a function that assigns an abstract event to each transition. For a path $\omega = (\hat{s}_1, \ldots, \hat{s}_n)$ through $\hat{\mathcal{P}}_G$ the* predicted abstract trace *is $\hat{\rho}_\omega := (\delta(\hat{s}_1, \hat{s}_2), \ldots, \delta(\hat{s}_{n-1}, \hat{s}_n))$.*

**Example** Let $\Sigma = \mathbb{N} \cup \{\perp\}$ be the set of concrete events and $\hat{\Sigma} = \mathbb{N} \times \mathbb{N}$ be the set of abstract events. An event $\sigma \in \Sigma$ represents the target of a memory accessing instruction. We define $\gamma(\hat{\sigma} = (n, m)) := \{i \mid n \leq i \leq m\}$. Furthermore let $\sigma_1 =$ 0x220, $\sigma_2 =$ 0x240, $\sigma_3 =$ 0x300 be concrete events, and $\hat{\sigma} =$ (0x200, 0x2ff) be an abstract event.

It holds $\hat{\sigma} \succ \sigma_1$ and $\hat{\sigma} \succ \sigma_2$ but $\hat{\sigma} \not\succ \sigma_3$. Hence the predicted abstract trace $\hat{\rho} = (\hat{\sigma}, \hat{\sigma})$ does not match the measured trace $\rho_1 = (\sigma_1, \sigma_3)$. But $\hat{\rho}$ matches the trace $\rho_2 = (\sigma_1, \sigma_2)$.

**Definition 5.4** (Predicted Abstract Trace Match)
*Let $\hat{\rho} = (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)$ be a predicted abstract trace and $\rho = (\sigma_1, \ldots, \sigma_m)$ be a concrete trace. The abstract trace $\hat{\rho}$* matches *the concrete trace, written as $\hat{\rho} \succ \rho$, iff $n = m$ and $\hat{\sigma}_i \succ \sigma_i$ for all $1 \leq i \leq n$.*

Given a perfect abstract hardware model and using a cycle-accurate measurement method any concrete trace should be matched by a corresponding predicted abstract trace. Otherwise, the abstract model of the analyzed processor is not sound. However, either due to unknown hardware behavior or due to limited measurement capabilities, a precise match between prediction and measurement is sometimes not possible. Thus we have to allow both local under- and overestimations as described in Definition 5.5.

**Definition 5.5** (Imprecise Predicted Abstract Trace Match)
*Let* $\hat{rho} = (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)$ *be a predicted abstract trace and* $\rho = (\sigma_1, \ldots, \sigma_m)$ *be a concrete trace. If* $\hat{\rho} \not\succ \rho$ *but* $\hat{\rho}$ *and* $\rho$ *can be partitioned into pairwise matching traces such that* $\hat{\rho} = \hat{\rho}_{\perp_1} \circ \hat{\rho}_1 \circ \hat{\rho}_{\perp_2} \circ \hat{\rho}_2 \circ \cdots \circ \hat{\rho}_n$ *and* $\rho = \perp^{j_1} \circ \rho_1 \circ \perp^{j_2} \circ \rho_2 \circ \cdots \circ \rho_n$ *where* $\hat{\rho}_{\perp_k} \succ \perp^{i_k}$ *and* $\hat{\rho}_k \succ \rho_k$ *with* $i_k, j_k \in \mathbb{N}$ *for any* $1 \leq k \leq n$*, the predicted abstract trace* $\hat{\rho}$ underestimates $\rho$ *if* $\sum_{k=1}^{n} i_k < \sum_{k=1}^{n} j_k$*, written as* $\hat{\rho} \precsim \rho$*, otherwise* $\hat{\rho}$ overestimates $\rho$*, written as* $\hat{\rho} \succsim \rho$*.*

The *prediction factor* is the quotient of the number of non-empty event transitions and the number of non-empty events in a trace. The factor provides a rough indication about the complexity of the analyzed hardware as well as of the input program. Both the complexity of the control-flow graph and the amount of splits that occur during abstract simulation contribute to a higher factor. Definition 5.6 provides a formal definition.

**Definition 5.6** (Prediction Factor)
*Let* $\Sigma$ *be a set of concrete events,* $\hat{\Sigma}$ *be a set of abstract events,* $\hat{\mathcal{P}}_G = (\hat{\mathcal{S}}, \hat{\mathcal{E}})$ *be a prediction graph, and* $\delta : \hat{\mathcal{E}} \to \hat{\Sigma}$ *a function that assigns an abstract event to each transition. The* prediction factor *is defined for concrete traces* $\rho = (\sigma_1, \ldots, \sigma_n)$ *for which there exists a predicted abstract trace* $\hat{\rho}$ *through* $\hat{\mathcal{P}}_G$ *such that* $\hat{\rho} \succsim \rho$ *or* $\hat{\rho} \precsim \rho$*.*

$$\varkappa(\rho) = \frac{|\{(\hat{s}, \hat{t}) \in \hat{\mathcal{E}} \mid \delta(\hat{s}, \hat{t}) \not\succ \perp\}|}{|\{i \mid \sigma_i \neq \perp\}|}$$

*If there exists a trace* $\rho'$ *with* $\varkappa(\rho') < 1$*, there exists no predicted abstract trace that either matches or overestimates the measurement. In this case, the abstract hardware model does not correctly model the hardware behavior.*

## 5.2 Measurement Granularity

A high trace resolution is desirable for an in-depth comparison between measurement and static WCET analysis. At which level of detail a validation can take place strongly depends on the available debug facilities of the processor. Some hardware

architectures allow for very fine-grained measurements that capture internal processor events, such as bus transaction signals or instruction dispatch. Other hardware only enables coarse traces. But a successful trace validation is still feasible with low-resolution measurements, even though the reason for a mismatch cannot be identified that easily.

We discern five different levels of granularity. We do not consider software-based instrumentation because the code modification does inevitably have a non-neglectable impact on the program timing behavior. A direct comparison between a static analysis of the unmodified program and measurements using the modified program is thus not possible. Hence, we focus on hardware monitoring as suggested in [31].

**end-to-end level** An in-depth analysis of the program or the hardware behavior is impossible. Any information about the program control-flow is lost. Program end-to-end times can only be used for a rough comparison between measurements and static analysis.

**per-routine level** This level of granularity allows for a better comparison between measurements and WCET analysis. Different routine execution contexts can be distinguished by reference to the call history. But it is not possible to gain precise information about the control-flow inside routines. Such measurement methods cannot provide detailed information about the processor behavior.

**per-block level** Measurement solutions that are able to extract the execution behavior per basic block and execution context provide much more data for trace validation. A widely spread debugging interface standard for embedded systems is IEEE-ISTO 5001-2003 Nexus [29]. Many Freescale PowerPC processors, such as the MPC5554 embedded micro-controller, support Nexus hardware traces.

However, due to limited resource capabilities and bandwidth[10], Nexus traces only record timestamps upon control-flow changes, e.g., due to taken conditional branches. Thus precise timestamps for each and every basic block cannot be obtained. Nonetheless it is possible to obtain the whole executed program control-flow that was subject to measurement. Per-block level traces provide much more insights into the processor's execution behavior than per-routine level measurements, but still leave enough room for guesswork.

**per-instruction level** Other, more advanced tracing solutions, like the Infineon multi-core debugging solution (MCDS) for the TriCore TC1796ED, provide cycle-exact timestamps for every executed instruction. The obtained data allows for full reconstruction of the program control-flow as well as for the time

[10] Nexus-conforming trace solutions need to transfer events upon occurrence directly over to measurement probes.

spent for instruction execution. Figure 5.2 shows an example per-instruction level measurement obtained by MCDS.

This level of detail is at least required to investigate the hardware behavior. However, to be able to fully understand how the processor executes an instruction a much higher resolution is necessary.

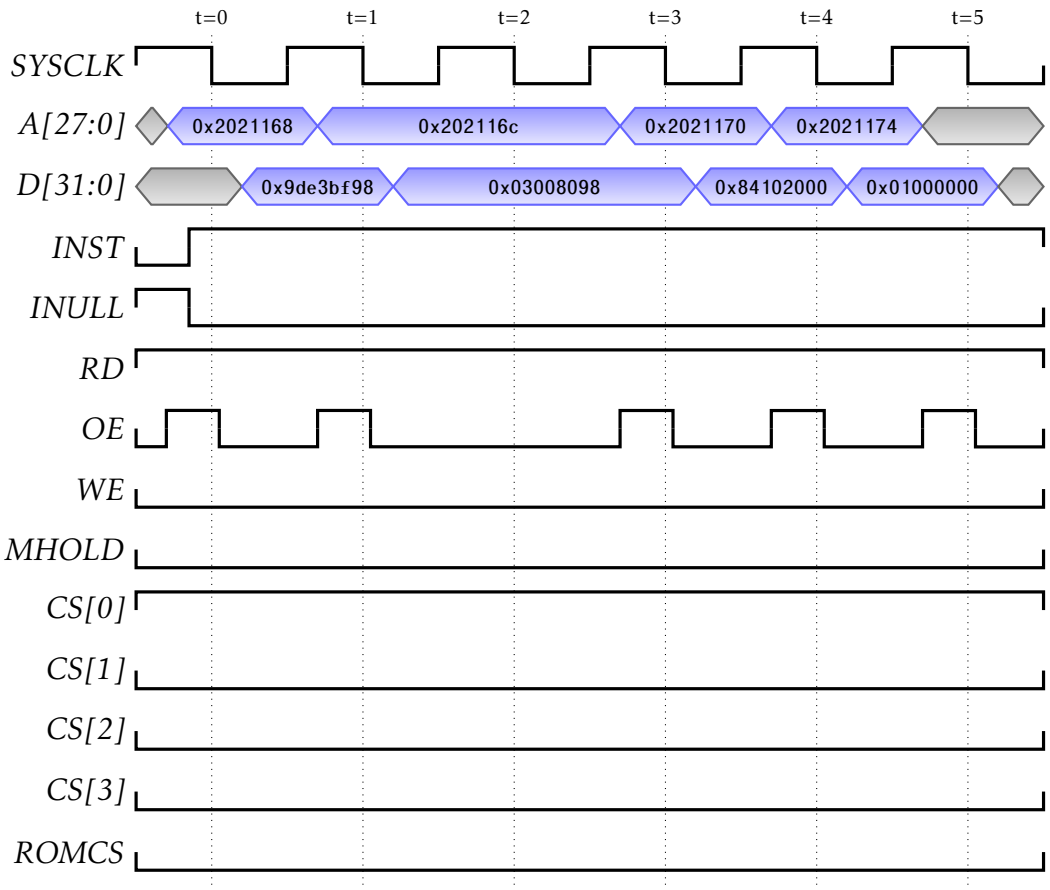
```
Tick      Address          Interpret
6         0xD4000034       CALL 0xd4000018
..        0xD4000018       JGE d4, 0x7, 0xd4000020
7         0xD400001C       ADD d15, d4, 0x7
8         0xD4000020       Prefetch: MUL d15, d4, 0x6
9         0xD400001E       J 0xd4000028
..        0xD4000028       ADD d2, d15, -0x3
10        0xD400002A       RET
```

Figure 5.2: *Example Per-Instruction Level Trace:* Instruction-level trace for a small TriCore program. Each processor tick represents the absolute number of cycles that have elapsed until the corresponding event has been recorded.

**per-action level** The full insight into the processor behavior is provided by per-action level traces. To enable this kind of measurement, the processor must provide access to internal signals, such as bus transfer signals like transfer start (TS), address acknowledge (AACK), or transfer acknowledge (TA). Some embedded processors even export pipeline-internal signals like the instruction dispatch, which allows for a very detailed view on the concrete hardware behavior.

Similarly, many modern embedded processors (e.g., Freescale MPC7448 or MPC5674F) offer performance counter registers that can be exploited to observe the execution behavior on a per-action level. Each of the performance counter registers can be configured to count several events, such as the number of dispatched or retired instructions. Depending on the processor complexity making use of performance counter registers to produce a trace is quite challenging. To understand the insights of a processor core, the traced time period must be chosen very carefully, and measurements need to be repeated several times to allow for tracing different events.

Per-action level measurements can be very costly. Every signal of interest needs to be connected to a measurement probe. The measurement hardware (i.e., usually a logic analyzer) needs to be able to cope with the processor's clock frequency. Such measurement hardware is very expensive for clock frequencies above 200 MHz. Figure 5.3 shows an excerpt of a logic analyzer measurement.




| Tick | Event | Type | Address |
|---|---|---|---|
| 1 | TS | Read | 0x02021168 |
| 2 | TS | Read | 0x0202116c |
| 4 | TS | Read | 0x02021170 |
| 5 | TS | Read | 0x02021174 |
| ... | | | |

Figure 5.3: *Example Per-Action Level Measurement:* Measurement excerpt for an ERC32 test program. The upper half of this figure shows the signals as recorded by the logic analyzer. Signals are extracted on falling clock edges as the dotted lines denote. The lower half of the figure depicts the observed per-action level trace. Each trace line denotes a new code fetch event. The transfer-acknowledge event is not part of this trace. The tick column indicates the number of elapsed processor cycles until the event has been recorded. In this particular example all but the third code fetch are initiated one after another.

## 5.3 Implementation

Based on Definition 5.4 on page 64 we can now provide an implementation of the trace validation algorithm. The proposed algorithm computes all paths through the prediction event graph that match or (globally) overestimate a given trace. Furthermore, the algorithm allows for a partial validation of the prediction event graph. A trace may only comprise events for subparts of the analyzed program. In any case, the trace validation is successful if the algorithm is able to determine a path through the graph that does not underestimate the input trace. In the following, we only consider traces that begin with a non-empty event $\sigma \neq \bot$.

Algorithm 5.1 provides a pseudo-code implementation. At first we search for edges to start the trace validation with. If the corresponding event matches the first event of the input trace, we insert a new item into the algorithm work list. A work list item comprises the path through the prediction graph we have considered so far, the distance between prediction and trace in terms of processor cycles, and the index of the last matched trace element. Given the initial work list the algorithm computes the paths that match or overestimate the input trace as follows.

First we remove the topmost work list item $(\pi = (\hat{s}_1, \ldots, \hat{s}_m), k, i)$. For every outgoing edge $(\hat{s}_m, \hat{t})$ we check whether the predicted event $\hat{\sigma} = \delta(\hat{s}_m, \hat{t})$ matches the next trace event $\sigma_{i+1}$. First we extend the path $\pi$ to $\pi' = (\hat{s}_1, \ldots, \hat{s}_m, \hat{t})$. There are two possibilities if $\hat{\sigma} \succ \sigma_{i+1}$. If we have matched all trace events, we add the match $(\pi', k)$ to the set of matches. Otherwise we insert the work list item $(\pi', k, i+1)$ to the work list. The distance between prediction and measurement remains untouched.

If $\hat{\sigma} \not\succ \sigma_{i+1}$, there are also two possibilities. We have found an *overestimation* if $\hat{\sigma} \succ \bot$. Later we continue with the work list item $(\pi', k+1, i)$ to attempt to match $\sigma_{i+1}$. The distance between trace and measurement increases. If $\sigma_{i+1} = \bot$, the prediction *underestimates* the measurement. The abstract event $\hat{\sigma}$ has been predicted too early if it eventually matches a succeeding trace event. Thus we insert the work list item $(\pi, k-1, i+1)$. The distance between trace and measurement decreases. If none of the three possibilities apply, the path $\pi$ cannot lead to a successful match of the input trace and is hence discarded.

In this fashion the algorithm continues until every path through the prediction graph has been visited that matches in accordance to the input trace. To exclude paths that globally underestimate the measurement, we exclude all matches $\mu = (\pi, k)$ where $k < 0$. If the final set of matches is not empty, the prediction event graph has been successfully validated for the input trace.

Figure 5.4 visualizes the output of the trace validation algorithm. Here the prediction graph contains a path that precisely matches the measured events (green path).

**Algorithm 5.1** Validation of Prediction Event Graph against Measured Trace

**function** TraceValidation
  **input** Prediction graph $\hat{\mathcal{P}}_G = (\hat{\mathcal{S}}, \hat{\mathcal{E}})$
    Abstract event per transition $\delta : \hat{\mathcal{E}} \to \hat{\Sigma}$
    Measured trace $\rho = (\sigma_1, \ldots, \sigma_n)$
  **output** Validation result *Matches*
  **begin**
    $Worklist \leftarrow []$ ▷ *Fill initial work list*
    **foreach** $(\hat{s}, \hat{t}) \in \hat{\mathcal{E}}$
      **if** $\delta(\hat{s}, \hat{t}) \succ \sigma_1$ **then**
        Push($Worklist$, $((\hat{s}, \hat{t}), 0, 1)$)
      **end if**
    **end foreach**

    $Matches \leftarrow \emptyset$
    **while** $Worklist \neq \emptyset$ **do**
      $(\pi = (\hat{s}_1, \ldots, \hat{s}_m), k, i) \leftarrow$Pop($Worklist$)
      **foreach** $(\hat{s}_m, \hat{t}) \in \hat{\mathcal{E}}$
        $\hat{\sigma} \leftarrow \delta(\hat{s}_m, \hat{t})$
        $\pi' \leftarrow (\hat{s}_1, \ldots, \hat{s}_m, \hat{t})$
        **if** $\hat{\sigma} \succ \sigma_{i+1}$ **then**
          **if** $i + 1 = n$ **then** ▷ *Finished match*
            $Matches \leftarrow Matches \cup \{(\pi', k)\}$
          **else** ▷ *Found match*
            Push($Worklist$, $(\pi', k, i + 1)$)
          **end if**
        **else if** $\hat{\sigma} \succ \perp$ **then** ▷ *Found local overestimation*
          Push($Worklist$, $(\pi', k + 1, i)$)
        **else if** $\sigma_{i+1} = \perp$ **then** ▷ *Found local underestimation*
          Push($Worklist$, $(\pi, k - 1, i + 1)$)
        **end if**
      **end foreach**
    **end while**

    **foreach** $\mu = (\pi, k) \in Matches$ ▷ *Forbid global underestimation*
      **if** $k < 0$ **then**
        $Matches \leftarrow Matches \backslash \{\mu\}$
      **end if**
    **end foreach**
  **end**
**end function**

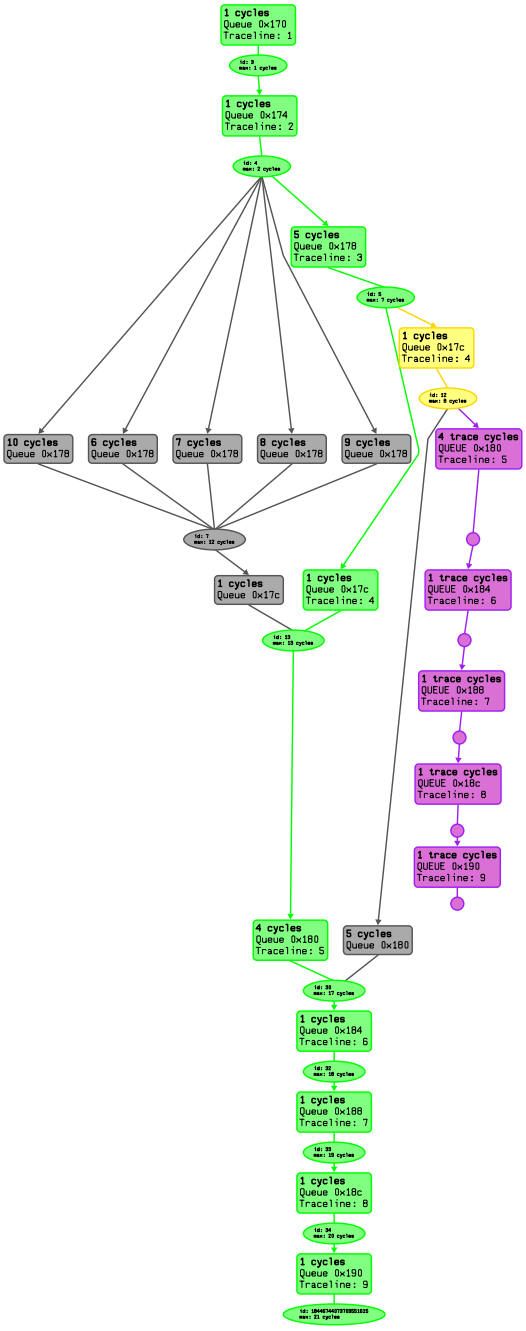



Figure 5.4: *Trace Validation Visualization:* The image depicts the output of the trace validation algorithm. The green path denotes a precise match of the input trace. Gray regions are parts of the prediction graph that do not match the measurement. Yellow edges denote matched events on over- or underestimated paths. The purple edges depict the unmatched parts of the input trace following over- or underestimated events.

Chapter 4 and Chapter 5 have presented means to validate static worst-case execution time analysis in the presence of timing anomalies. Section 6.1 demonstrates the trace validation of the static WCET analyzer aiT for various hardware architectures. The validation of aiT is exemplified on custom benchmark programs and on avionics applications. Section 6.2 investigates and visualizes found timing anomalies.

# 6.1 Trace Validation

## 6.1.1 ERC32

The ERC32 processor is a SPARC v7 compliant hardware architecture. It consists of three modules comprising an integer unit (IU), a floating-point unit (FPU), and a memory controller (MEC). Figure 6.1 depicts the ERC32 pipeline model that has been derived from a behavioral VHDL model [10].

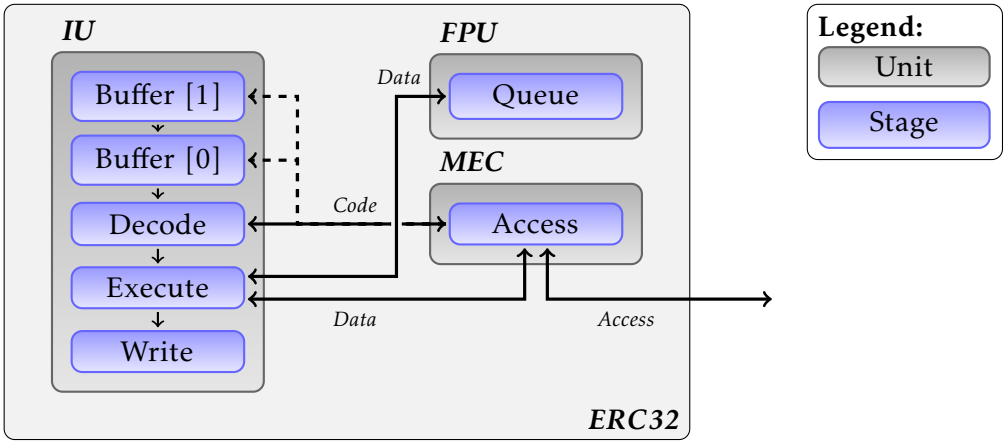



Figure 6.1: *ERC32 Pipeline Model:* The model of the ERC32 pipeline comprises three communicating units: the integer unit (IU), the floating-point unit (FPU), and the memory controller (MEC). The edges denote the data paths between the units.

### Hardware Description

The IU employs a four-stage instruction pipeline that permits parallel execution of multiple instructions. According to the VHDL model there are two prefetch buffers

that allow the IU to prefetch subsequent instructions while executing instructions that occupy multiple pipeline stages.

The IU fully controls the FPU, fetches FPU instructions and performs load and stores of floating-point data. By means of a one-entry floating-point instruction queue, which stores the currently executed operation, the FPU can execute one FPU instruction in parallel to the IU. If an additional floating-point instruction is to be dispatched while a previous instruction occupies the FPU queue, the IU is stalled until the previous instruction has finished execution. The pipeline model only comprises the floating-point queue, because FPU instructions pass the IU and FPU stages in parallel.

The MEC provides the interface to internal and external memory resources and is fully controlled through the IU. The model supports all internal memories, such as PROM, extended PROM, exchange RAM, SRAM, extended SRAM, and I/O memory regions. Each of the memories can be configured separately with static read and write waitstate timing. The port width of the internal PROM can be additionally configured to 32 bit.

### Measurement Setup

For our experiments the European Space Agency (ESA) kindly provided us with a Saab Ericsson Space TSC695F Compact PCI board. The TSC695F board is a single-chip implementation of the ERC32 processor. The processor is running at 20 MHz. The board comprises 512 KB boot EEPROM (8 bit port width) and 8 MB SRAM (32 bit port width) with error-detection-and-correction (EDAC) protection.

The ERC32 board does not possess any debug facilities that allow us to directly observe the internals of the processor pipeline. Fortunately, the board features six debug connectors that provide a full pin-out of the internal memory bus. This allows us to observe the processor behavior on a per-action level. By means of a logic analyzer, we record the following signals (see Figure 5.3 on page 68 as an example).

**SYSCLK** The system clock (SYSCLK) is generated by the ERC32 for clocking the IU and the FPU as well as other system logic.

**A[27:0]** The address bus for the ERC32 is an output bus. Inside the processor, the IU address bus is used to perform decoding, to generate select signals and to check against the memory access protection scheme. It is also used to address the system registers.

The upper address bits A[31:28] are not made available via the debug connectors. This still allows for a successful comparison because the address bus cov-

ers the address range where the test programs are linked to (i.e., 0x02000000 ... 0x027fffff).

**D[31:0]** These signals form a 32-bit bidirectional data bus that serves as the interface between the ERC32 and external memory. The data bus is not driven by the ERC32 during system register accesses, it is only driven during the execution of integer and floating-point load and store instructions and the store cycle of atomic-load-store instructions on external memory.

For comparison purposes measuring the data bus signals are not essential. We have included the data bus to cross-check the data with the operation code of the assembly instructions.

**INST** The instruction fetch signal (INST) is asserted by the IU whenever a new instruction is being fetched. It is used by the FPU to latch the instruction currently on the internal data bus into an FPU instruction buffer.

**INULL** The processor asserts the integer unit nullify signal (INULL) to indicate that the current memory access is being nullified. It is asserted at the beginning of the cycle in which the address being nullified is active.

**RD** The read access signal (RD) is sent out during the address portion of an access to specify whether the current memory access is a read (RD = 1) or a write (RD = 0) operation. RD is set low only during the address cycles of store instructions. For atomic load-store instructions, RD is set high during the load address cycle and set low during the two store address cycles.

**OE** The output enable signal (OE) is asserted during fetch or load accesses to the main memory. It is intended to be used to control memory devices with output enable features.

**WE** The write enable signal (WE) is asserted by the IU during the cycle in which the store data is on the data bus. For a store single instruction, this is during the second store address cycle, the second and third store address cycles of store double instructions and the third load-store address cycle of atomic load-store instructions.

**MHOLD** This signal is, among others, asserted when a *Memory Hold* (MHOLD) or a *Floating-point Hold* (FHOLD) is internally generated. MHOLD is used to freeze the pipeline to both the IU and FPU accessing a slow memory or during memory exception. The IU and FPU internal outputs return to and stay at the value they had on the rising edge of SYSCLK in the cycle in which MHOLD was asserted. FHOLD is asserted by the FPU if a situation arises in which the FPU cannot continue execution. The FPU checks all dependencies in the decode stage of the instruction and asserts an FHOLD in the next cycle.

**ROMCS** The ROM chip select output (ROMCS) is asserted whenever there is an access to the boot ROM area.

**CS[3:0]** The memory chip select signals are asserted during an access to the main memory.

### Results

For each hardware feature we design a dedicated test case in assembly in order to cover the whole functionality of the ERC32 processor. This includes tests for all available FPU instructions. In total we have implemented 75 different tests programs. 43 of the test programs are related to the floating-point unit.

All tests successfully pass the trace validation. Table 6.1 shows the trace validation result for an excerpt of the test set. We observe that the ERC32 pipeline analysis is cycle-accurate except for the floating-point tests. For each FPU instruction we have implemented a dedicated test to measure the FPU execution behavior in isolation on randomly generated floating-point numbers. The overestimation for these tests is inherently high because most floating-point instructions do not complete after a fixed number of cycles. The worst-case execution behavior is seldom observed. For example, the `fadds` instruction finishes after 4 to 17 cycles. In most cases however, the instruction completes after 4 cycles only. Unfortunately, we did not have access to the concrete VHDL model, which would allow for a more precise static WCET analysis.

Table 6.2 lists the number of measurement events compared to the prediction graph size (i.e., number of nodes, number of edges, and number of predicted events). For the FPU tests, the prediction graph size is significantly larger because these tests execute a corresponding floating-point instruction in a loop. We have measured up to 1000 execution instances for each FPU instruction.

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| annulled-load | 12 | 12 | 0.00 |
| annulled-load-nottaken | 14 | 14 | 0.00 |
| annulled-load-taken | 27 | 27 | 0.00 |
| branch | 25 | 25 | 0.00 |
| branch-dep | 30 | 30 | 0.00 |
| branch-store | 16 | 16 | 0.00 |
| call | 38 | 38 | 0.00 |
| call-dep | 20 | 20 | 0.00 |
| call-load-prom | 72 | 72 | 0.00 |
| call-load-sram | 46 | 46 | 0.00 |
| call-nop | 44 | 44 | 0.00 |
| call-store | 18 | 18 | 0.00 |
| fpu-abss | 4006 | 4007 | 0.02 |
| fpu-dtoi | 3072 | 3465 | 12.79 |
| fpu-dtos | 3527 | 3918 | 11.09 |
| fpu-fsr | 300 | 323 | 7.67 |
| fpu-movs | 7806 | 7807 | 0.01 |
| fpu-negs | 4006 | 4007 | 0.02 |
| fpu-sqrtd | 9142 | 11133 | 21.78 |
| fpu-sqrts | 6353 | 7349 | 15.68 |
| fpu-stod | 2006 | 2405 | 19.89 |
| fpu-stoi | 2674 | 3400 | 27.15 |
| load-jmpl | 21 | 21 | 0.00 |
| load-prom | 127 | 127 | 0.00 |
| load-sram | 23 | 23 | 0.00 |
| store-load-dep | 249 | 249 | 0.00 |
| store-sram-8bit | 55 | 55 | 0.00 |
| store-sram-16bit | 55 | 55 | 0.00 |
| store-sram-32bit | 31 | 31 | 0.00 |
| store-sram-64bit | 27 | 27 | 0.00 |

Table 6.1: *ERC32 Analysis Results:* Trace validation results for the static WCET analyzer aiT for ERC32. Due to the lack of precise information about the timing behavior of FPU instructions, the analysis is not able to precisely predict the measured behavior. At worst, we observe a difference to the highest measured execution time of 27.15%.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| annulled-load | 11 | 23 | 22 | 15 | (x1.36) |
| annulled-load-nottaken | 13 | 25 | 24 | 17 | (x1.31) |
| annulled-load-taken | 12 | 24 | 23 | 16 | (x1.33) |
| branch | 24 | 63 | 69 | 55 | (x2.29) |
| branch-dep | 12 | 34 | 34 | 24 | (x2.00) |
| branch-store | 13 | 26 | 25 | 17 | (x1.31) |
| call | 32 | 69 | 68 | 36 | (x1.12) |
| call-dep | 14 | 33 | 32 | 18 | (x1.29) |
| call-load-prom | 38 | 75 | 74 | 42 | (x1.11) |
| call-load-sram | 38 | 75 | 74 | 42 | (x1.11) |
| call-nop | 38 | 75 | 74 | 42 | (x1.11) |
| call-store | 14 | 32 | 31 | 18 | (x1.29) |
| fpu-abss | 3606 | 5218 | 5217 | 3610 | (x1.00) |
| fpu-dtoi | 1806 | 22403 | 23809 | 14396 | (x7.97) |
| fpu-dtos | 1806 | 40393 | 43403 | 27585 | (x15.27) |
| fpu-fsr | 240 | 1724 | 1840 | 1518 | (x6.33) |
| fpu-movs | 7206 | 10218 | 10217 | 7210 | (x1.00) |
| fpu-negs | 3606 | 5218 | 5217 | 3610 | (x1.00) |
| fpu-sqrtd | 1806 | 209669 | 224541 | 134862 | (x74.67) |
| fpu-sqrts | 1806 | 132727 | 142370 | 86919 | (x48.13) |
| fpu-stod | 1806 | 30201 | 32607 | 21393 | (x11.85) |
| fpu-stoi | 1806 | 22203 | 23608 | 14396 | (x7.97) |
| load-jmpl | 15 | 29 | 28 | 19 | (x1.27) |
| load-prom | 14 | 26 | 25 | 18 | (x1.29) |
| load-sram | 14 | 26 | 25 | 18 | (x1.29) |
| store-load-dep | 88 | 150 | 149 | 92 | (x1.05) |
| store-sram-8bit | 14 | 34 | 33 | 18 | (x1.29) |
| store-sram-16bit | 14 | 34 | 33 | 18 | (x1.29) |
| store-sram-32bit | 14 | 34 | 33 | 18 | (x1.29) |
| store-sram-64bit | 10 | 26 | 25 | 14 | (x1.40) |

Table 6.2: *ERC32 Test Complexity:* Number of measured events and corresponding prediction graph size. Contrary to the non-floating-point tests, the prediction graph size of the FPU tests is significantly larger. The large variability of the FPU instruction timing and the code structure of these tests contribute to the size of the prediction graph.

## 6.1.2 LEON2

The LEON2 processor is a SPARC v8 compliant hardware architecture. It is a follow-up design of the ERC32 processor. The instruction pipeline comprises an additional execute stage that is dedicated to memory accesses only. Furthermore, the LEON2 features disjoint instruction and data caches. Figure 6.2 depicts the LEON2 pipeline model that has been derived from a behavioral VHDL model.

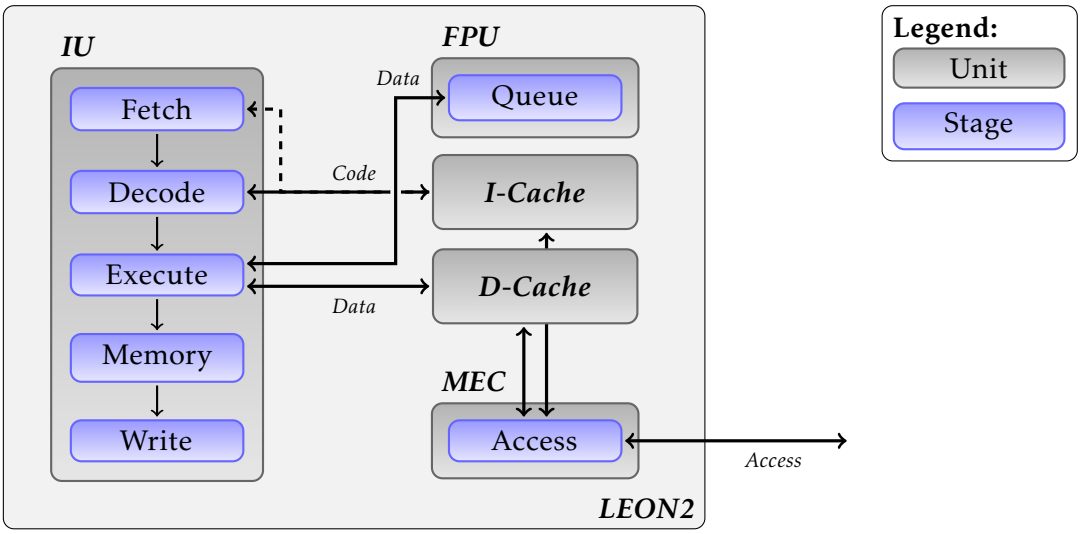



Figure 6.2: *LEON2 Pipeline Model:* The model of the LEON2 pipeline comprises five communicating units: the integer unit (IU), the floating-point unit (FPU), the memory controller (MEC), and the instruction and data caches. The edges denote the data paths between the units.

### Hardware Description

The integer unit employs a five-stage in-order instruction pipeline to execute multiple instructions in parallel. The fetch stage (FE) requests the next instruction from the instruction cache, or forwards its request directly to the memory controller if the instruction cache is disabled. The decode stage (DE) decodes the instruction and identifies the read operands either from the register file or from internal data bypasses. Additionally, call and branch target addresses are generated in this stage. The execute stage (EX) performs arithmetic-logic instructions and computes target addresses for memory accessing instructions. The memory stage (ME) advances pending requests to the data cache, or forwards the request directly to the memory controller. The write stage (WR) updates the register file.

For the LEON2 processor there are two IEEE-754-compliant FPU implementations available that operate on both single- and double-precision operands, and imple-

ment all SPARC V8 FPU instructions. The Meiko FPU operates in serial fashion, where floating-point instruction do not execute in parallel with IU instruction and the processor is stopped during FPU execution. The Gaisler Research floating-point unit (GRFPU) is interfaced with the integer unit that allows the FPU to execute instructions in parallel to the integer pipeline. Only in case of dependencies the LEON2 pipeline waits. The model only supports the Meiko implementation.

The MEC provides the interface to internal and external memory resources and is fully controlled through the IU. The model supports all internal memories, such as PROM, SRAM, SDRAM, and I/O memory regions. Each of the memories can be configured separately with static read and write waitstate timing. The refresh of the SDRAM is not modeled and has to be considered separately. The port width of each memory module, except for SDRAM, can be configured to 8 bit, 16 bit or 32 bit.

### Measurement Setup

For our experiments we were provided a GR-CPCI-AT697 board that runs a LEON2 core with up to 100 MHz. That particular LEON2 processor features a 4-way set-associative 32 KB instruction cache and a 2-way set-associative 16 KB data cache. Both caches are updated according to the LRU cache replacement policy. The board comprises 16 MB internal PROM (32 bit port width), 1 MB internal SRAM (32 bit port width), and a 512 MB SDRAM module (32 bit port width). Both SRAM and SDRAM are equipped with error-detection-and-correction (EDAC) protection.

Similar to the ERC32 (see Section 6.1.1 on page 73), the LEON2 board does not possess debug facilities to observe the integer pipeline. However, the available LEON2 evaluation board provides a full pin-out of the internal memory bus. This allows us to observe the processor behavior on a per-action level. By means of a logic analyzer, we obtain an event trace by recording the following signals.

**CLK** The processor clock (CLK) provides the main reference for the processor.

**A[27:0]** The address bus carries the addresses during accesses to external memory. When access to cache memory is performed, the address of the last external memory access remains driven on the address bus. The upper address bits A[31:28] are not made available.

For the measurements we only used the lower 16 bits of the address bus. This still allows for a successful comparison due to the rather small size of the test programs.

**D[31:0]** The bi-directional data bus carries the data during accesses to memory. The processor automatically configures the bus as output and drives the lines during write cycles. During accesses to 8-bit areas, only D[31:24] are used. During accesses to 16-bit areas, only D[31:16] are used.

For comparison purposes measuring the data bus signals are not essential. We have included the data bus to cross-check the data with the operation code of the assembly instructions.

**READ** The LEON2 processor asserts the read signal during read cycles on the memory bus.

**WRITE** The write signal provides a write strobe during write cycles on the memory bus.

**ROMS[1:0]** The ROMS output signals provide the chip-select signal for the PROM area. ROMS[0] is asserted once the lower half of the PROM area is accessed (i.e., 0x0 … 0x0fffffff), whereas ROMS[1] is asserted for the upper half.

**RAMS[4:0]** The RAMS signals provide the chip-select signals for each RAM bank. We use these signals to identify when an access to the internal SRAM memory occurs.

### Results

To cover the whole functionality of the LEON2 processor we have implemented 112 test cases. All tests successfully pass the trace validation. Table 6.3 shows the trace validation results for an excerpt of the test set. The results show that LEON2 pipeline analysis is cycle-accurate, except for the FPU and the caches.

The precision of the cache analysis is tightly coupled to the structure of the code and the hardware configuration. In general, the cache analysis has no knowledge about the contents of the caches. Assuming unknown contents together with the instruction cache configured to burst-fill cache lines, might cause the cache analysis to predict pipeline states that would not occur in reality. No effort was spent to effectuate a bad/worst-case cache filling for the execution time measurements. The possibility of a timing anomaly (as discussed in Section 4.5 on page 56) further contributes to this problem. For these tests, we observe a difference to the highest measured execution time of up to 1.54%.

By mischance, there was no precise documentation about the Meiko FPU available and thus no information about possible early-out mechanisms present (or similar techniques to speed up the execution of FPU instructions). But contrary to the ERC32 floating-point unit, the Meiko FPU instruction timing is less volatile. Hence, we recognize a difference to the highest measured execution time of up to 9.21% (on average about 4%) for the FPU instruction test cases.

Table 6.4 lists the number of measurement events compared to the prediction graph size (i.e., number of nodes, number of edges, and number of predicted events).

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| annulled | 51 | 51 | 0.00 |
| annulled-load | 81 | 81 | 0.00 |
| branch-cached | 137 | 137 | 0.00 |
| call | 156 | 156 | 0.00 |
| call-cached | 418 | 420 | 0.48 |
| div | 546 | 546 | 0.00 |
| dry2_1 | 11438 | 11439 | 0.01 |
| interlock-mul | 73 | 73 | 0.00 |
| interlock-store | 179 | 179 | 0.00 |
| fpu-add | 8021 | 8623 | 7.51 |
| fpu-cmp | 12619 | 12786 | 1.32 |
| fpu-div | 8722 | 9525 | 9.21 |
| fpu-itod | 10206 | 10482 | 2.70 |
| fpu-mul | 11622 | 11623 | 0.01 |
| fpu-sqrt | 10922 | 10975 | 0.49 |
| fpu-sub | 8520 | 8625 | 1.23 |
| icache-fill | 61 | 62 | 1.64 |
| icache-fill-hit | 65 | 65 | 0.00 |
| jumpl | 83 | 83 | 0.00 |
| load-cached | 114 | 114 | 0.00 |
| load-double-cached | 431 | 431 | 0.00 |
| load-prom | 292 | 292 | 0.00 |
| load-prom-cached | 324 | 329 | 1.54 |
| load-prom-call | 373 | 373 | 0.00 |
| load-sram | 81 | 82 | 1.23 |
| mul | 256 | 256 | 0.00 |
| store-double-single | 119 | 119 | 0.00 |
| store-single | 112 | 112 | 0.00 |
| store-sram | 170 | 170 | 0.00 |
| store-sram-rmw | 289 | 289 | 0.00 |

Table 6.3: *LEON2 Analysis Results:* Trace validation results for the static WCET analyzer aiT for LEON2. Due to the lack of precise information about the timing behavior of FPU instructions, the analysis is not able to precisely predict the measured behavior. At worst, we observe a difference to the highest measured execution time of 9.21%.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| annulled | 11 | 24 | 23 | 11 | (x1.00) |
| annulled-load | 11 | 24 | 23 | 11 | (x1.00) |
| branch-cached | 51 | 293 | 362 | 182 | (x3.57) |
| call | 31 | 63 | 62 | 32 | (x1.03) |
| call-cached | 153 | 1482 | 1762 | 943 | (x6.16) |
| div | 42 | 107 | 122 | 61 | (x1.45) |
| dry2_1 | 1729 | 2887 | 2886 | 1729 | (x1.00) |
| interlock-mul | 8 | 36 | 42 | 14 | (x1.75) |
| interlock-store | 17 | 35 | 34 | 17 | (x1.00) |
| fpu-add | 1521 | 8971 | 12585 | 4837 | (x3.18) |
| fpu-cmp | 2129 | 4861 | 5756 | 2898 | (x1.36) |
| fpu-div | 1521 | 12671 | 19185 | 5037 | (x3.31) |
| fpu-itod | 1467 | 9529 | 14078 | 3118 | (x2.13) |
| fpu-mul | 2121 | 3071 | 3085 | 2137 | (x1.01) |
| fpu-sqrt | 821 | 1371 | 1385 | 837 | (x1.02) |
| fpu-sub | 1521 | 8971 | 12585 | 4837 | (x3.18) |
| icache-fill | 25 | 171 | 208 | 119 | (x4.76) |
| icache-fill-hit | 25 | 175 | 206 | 99 | (x3.96) |
| jumpl | 17 | 26 | 25 | 18 | (x1.06) |
| load-cached | 17 | 34 | 34 | 19 | (x1.12) |
| load-double-cached | 34 | 67 | 71 | 46 | (x1.35) |
| load-prom | 13 | 28 | 27 | 14 | (x1.08) |
| load-prom-cached | 21 | 122 | 142 | 76 | (x3.62) |
| load-prom-call | 38 | 91 | 90 | 38 | (x1.00) |
| load-sram | 13 | 28 | 27 | 14 | (x1.08) |
| mul | 42 | 107 | 122 | 61 | (x1.45) |
| store-double-single | 14 | 30 | 29 | 14 | (x1.00) |
| store-single | 14 | 30 | 29 | 14 | (x1.00) |
| store-sram | 29 | 47 | 46 | 30 | (x1.03) |
| store-sram-rmw | 30 | 59 | 58 | 30 | (x1.00) |

Table 6.4: *LEON2 Test Complexity:* Number of measured events and corresponding prediction graph size. The prediction graph size increases with a higher variability of the processor's execution behavior. This especially applies to tests related to the caches and to the floating-point unit.

### 6.1.3 M68020

The Motorola M68020 processor was the first full 32 bit implementation of the M68000 microprocessor family. The processor is equipped with an M68882 floating-point co-processor that is fully IEEE-754 floating-point compatible. Figure 6.3 depicts the manually derived M68020/M68882 pipeline model.

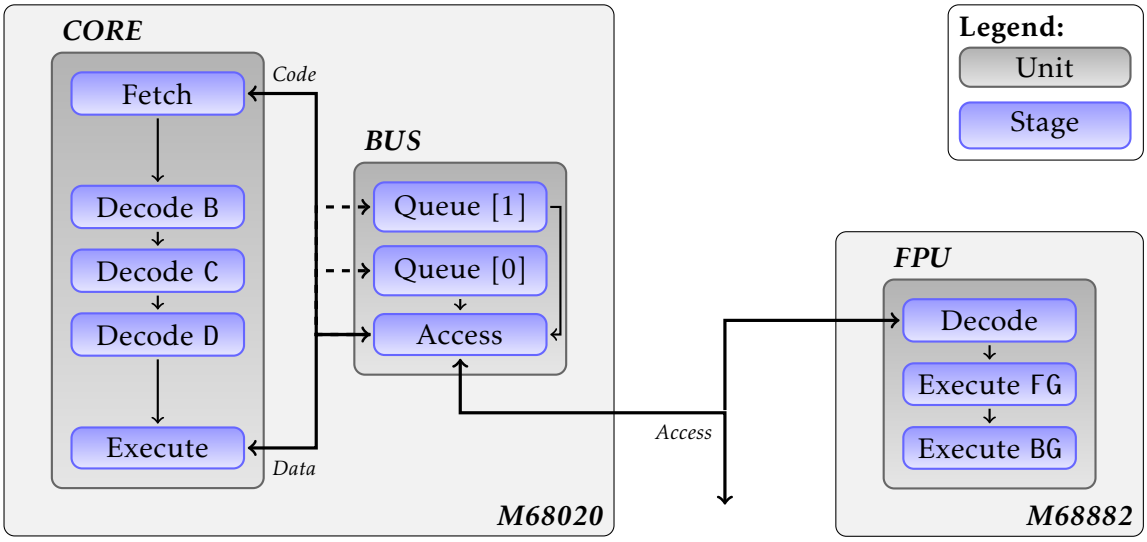



Figure 6.3: *M68020 Pipeline Model:* The pipeline model comprises the M68020 processor and the M68882 floating-point co-processor. The M68020 consists of an integer instruction pipeline (CORE) and a bus controller (BUS). The bus controller establishes the communication between memory and the FPU co-processor. The M68882 comprises a three stage pipeline that allows for the concurrent execution of up to two FPU instructions in parallel to the M68020 core.

#### Hardware Description

The M68020 is a 32 bit CISC processor that is fully compatible to earlier M68000 microprocessors. Other than the previously discussed architectures, the M68000 instruction set architecture uses a variable-length instruction encoding supporting up to 16 byte wide instructions.

The processor employs a three-stage pipeline. The fetch stage holds up to 4 bytes of code. The decode stage is subdivided into three sub-stages (`B`, `C`, and `D`) that process fetched data on chunks of 2 bytes each. While executing an instruction, the decode stages feed the corresponding functional units with operand data.

The bus controller establishes the communication between processor, memories and other peripherals. The bus interface allows operands to be located in memory on any

byte boundary and provides access to 8 bit, 16 bit or 32 bit port sized memories or peripheral devices. While serving an access the bus controller allows two additional outstanding requests.

The M68882 floating-point unit is controlled by the M68020 processor via the bus interconnect. The processor is in full control of the FPU and steers it by means of dedicated memory accesses. Hereby, the target address encodes the operation to execute, e.g., a memory write access to 0x2200a passes a 16 bit operand to the FPU. FPU instructions are either executed in foreground (FG) or in background (BG), allowing the FPU to execute another instruction in foreground. An FPU instruction is kept in foreground if it awaits input data from the processor. In total two FPU instructions can be executed in parallel.

The M68882 floating-point unit is rather slow. Many instructions take hundreds of processor cycles to complete execution. In addition the execution time variance of FPU instructions is rather high and strongly depends on the operands. For example, a floating-point addition may take 51 to 78 cycles to complete.

#### Measurement Setup

For our experiments we used an M68020/M68882 evaluation board that is clocked at 20 MHz. The pin-out of the board provides full access to the internal memory bus. This allows for an in-depth view on the communication of the M68020 processor between the M68882 co-processor and the connected peripherals. Debug access to the processor pipeline is not available. By means of a logic analyzer, we record the following signals.

**CLK** The clock signal (CLK) is the clock input to the M68020.

**A[31:0]** The signals provide the address for the current bus cycle. The communication with the M68882 co-processor can also be observed.

**D[31:0]** The bi-directional data bus provides the general-purpose data path between the processor and the connected peripherals. The data bus transfers up to 32 bits of data per bus cycle. The control commands for the M68882 FPU are also transmitted over the data bus.

**DSACK0, DSACK1** These input signals indicate the completion of a requested data transfer operation. In addition, they indicate the size of the external bus port at the completion of each cycle.

**SIZ0, SIZ1** The signals indicate the number of bytes remaining to be transferred for the current bus cycle. The signals A1, A0, DSACK0, DSACK1, SIZ0, and SIZ1 define the number of bits on the data bus.

**ECS** The output signal ECS denotes the beginning of a bus cycle of any type. We interpret ECS as a transfer-start signal.

**R/W** The read-write signal determines the bus access type.

**AS** The address strobe signal (AS) indicates that a valid address is latched on the address bus.

### Results

We obtained the results from a trace validation of a real-word avionics application running on an M68020/M68882 processor. For our experiments we had access to two different implementations of the application, one of which uses the M68882 co-processor and the other one uses an external processor for floating-point computation. The whole application follows a triple modular redundant design. By means of a shared bus the application instances match their (intermediate) results against each other (voting system).

Both implementations of the application successfully pass the trace validation. In total over 1.5 million lines of trace comprising about 1.3 million bus events were automatically successfully compared against the M68020/M68882 pipeline model. Table 6.5 depicts an excerpt of the trace validation results for both implementations of the avionics application. The tasks of the two implementations cannot be directly related to each other because the difference in their implementation is too huge. Due to NDA agreements we must not provide the real task names.

The application under investigation is highly data-dependent. As expected the difference between the observed execution time and the predicted worst-case execution time bound for the avionic$_{fpu}$ tasks is much higher than for the tasks using an external processor for FPU computation. On average we observe a difference of 20.36%. This is caused by the high variance in the execution behavior of the M68882 FPU. Unfortunately the static worst-case analysis has no insights in the internal functioning of this FPU. For the non-M68882 FPU task set, we observe a difference of 6.06% on average, which is acceptable for data-dependent software.

Table 6.6 shows the prediction graph size for each of the selected tasks. The static WCET analysis may always assume the worst execution time for each FPU instruction because the M68882 co-processor stalls if both FPU execution stages are occupied. Hence, we do not observe a major difference of the prediction graph complexity between the two implementations.

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| $\text{avionic}_1$ | 522 | 549 | 5.17 |
| $\text{avionic}_2$ | 1326 | 1415 | 6.71 |
| $\text{avionic}_3$ | 2061 | 2217 | 7.57 |
| $\text{avionic}_4$ | 358 | 374 | 4.47 |
| $\text{avionic}_5$ | 1369 | 1451 | 5.99 |
| $\text{avionic}_6$ | 10117 | 10681 | 5.57 |
| $\text{avionic}_7$ | 10768 | 11420 | 6.05 |
| $\text{avionic}_8$ | 10128 | 10688 | 5.53 |
| $\text{avionic}_9$ | 1507 | 1592 | 5.64 |
| $\text{avionic}_{10}$ | 7677 | 8333 | 8.55 |
| $\text{avionic}_{11}$ | 3816 | 3989 | 4.53 |
| $\text{avionic}_{12}$ | 1859 | 1923 | 3.44 |
| $\text{avionic}_{13}$ | 6421 | 6610 | 2.94 |
| $\text{avionic}_{14}$ | 8351 | 9115 | 9.15 |
| $\text{avionic}_{15}$ | 1060 | 1162 | 9.62 |
| $\text{avionic}_{fpu_1}$ | 10179 | 12067 | 18.55 |
| $\text{avionic}_{fpu_2}$ | 8593 | 10809 | 25.79 |
| $\text{avionic}_{fpu_3}$ | 8068 | 9988 | 23.80 |
| $\text{avionic}_{fpu_4}$ | 548 | 650 | 18.61 |
| $\text{avionic}_{fpu_5}$ | 8540 | 10205 | 19.50 |
| $\text{avionic}_{fpu_6}$ | 2230 | 2830 | 26.91 |
| $\text{avionic}_{fpu_7}$ | 4495 | 5332 | 18.62 |
| $\text{avionic}_{fpu_8}$ | 4195 | 5119 | 22.03 |
| $\text{avionic}_{fpu_9}$ | 2943 | 3330 | 13.15 |
| $\text{avionic}_{fpu_{10}}$ | 10710 | 12801 | 19.52 |
| $\text{avionic}_{fpu_{11}}$ | 3195 | 3785 | 18.47 |
| $\text{avionic}_{fpu_{12}}$ | 430 | 504 | 17.21 |
| $\text{avionic}_{fpu_{13}}$ | 3994 | 4709 | 17.90 |
| $\text{avionic}_{fpu_{14}}$ | 3693 | 4405 | 19.28 |
| $\text{avionic}_{fpu_{15}}$ | 2971 | 3748 | 26.15 |

Table 6.5: *M68020 Analysis Results:* Trace validation for an avionics application running on an M68020/M68882 processor. The variance in FPU execution timing causes a higher difference between measurement and prediction. For the $\text{avionic}_{fpu}$ we recognize at worst a difference of 26.91%.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| $avionic_1$ | 88 | 100 | 99 | 88 | (x1.00) |
| $avionic_2$ | 526 | 5583 | 5756 | 4420 | (x8.40) |
| $avionic_3$ | 817 | 7109 | 7448 | 5600 | (x6.85) |
| $avionic_4$ | 172 | 472 | 486 | 377 | (x2.19) |
| $avionic_5$ | 543 | 4776 | 4931 | 3616 | (x6.66) |
| $avionic_6$ | 3760 | 20549 | 21586 | 14944 | (x3.97) |
| $avionic_7$ | 4078 | 11182 | 11762 | 8385 | (x2.06) |
| $avionic_8$ | 3762 | 20556 | 21593 | 14950 | (x3.93) |
| $avionic_9$ | 594 | 3629 | 3733 | 2821 | (x4.75) |
| $avionic_{10}$ | 2674 | 21956 | 22938 | 16517 | (x6.24) |
| $avionic_{11}$ | 1324 | 3113 | 3252 | 2245 | (x1.70) |
| $avionic_{12}$ | 676 | 1595 | 1679 | 1116 | (x1.65) |
| $avionic_{13}$ | 2422 | 6738 | 6838 | 5429 | (x2.24) |
| $avionic_{14}$ | 2846 | 21949 | 22931 | 16511 | (x5.80) |
| $avionic_{15}$ | 286 | 558 | 569 | 427 | (x1.49) |
| $avionic_{fpu_1}$ | 2710 | 28593 | 29691 | 19427 | (x7.17) |
| $avionic_{fpu_2}$ | 1764 | 4569 | 4625 | 3873 | (x2.20) |
| $avionic_{fpu_3}$ | 1592 | 5741 | 5839 | 4519 | (x2.84) |
| $avionic_{fpu_4}$ | 118 | 157 | 157 | 132 | (x1.12) |
| $avionic_{fpu_5}$ | 2110 | 17844 | 18433 | 12424 | (x5.58) |
| $avionic_{fpu_6}$ | 546 | 2120 | 2175 | 1589 | (x2.91) |
| $avionic_{fpu_7}$ | 1028 | 2401 | 2445 | 1979 | (x1.93) |
| $avionic_{fpu_8}$ | 848 | 1205 | 1215 | 1032 | (x1.22) |
| $avionic_{fpu_9}$ | 900 | 2588 | 2655 | 2050 | (x2.28) |
| $avionic_{fpu_{10}}$ | 2634 | 15797 | 16338 | 11185 | (x4.25) |
| $avionic_{fpu_{11}}$ | 870 | 2379 | 2448 | 1875 | (x2.16) |
| $avionic_{fpu_{12}}$ | 168 | 472 | 486 | 377 | (x2.24) |
| $avionic_{fpu_{13}}$ | 930 | 2658 | 2716 | 2211 | (x2.38) |
| $avionic_{fpu_{14}}$ | 810 | 1643 | 1661 | 1383 | (x1.71) |
| $avionic_{fpu_{15}}$ | 732 | 2717 | 2790 | 1941 | (x2.65) |

Table 6.6: *M68020 Analysis Complexity:* Number of measured events compared to the events predicted by the static WCET analyzer. There is no significant difference between the two implementations of the avionic software, as the analysis assumes the worst execution time for each FPU instruction.

### 6.1.4 MPC5xx

The Freescale MPC5xx family of processors consist of 32 bit PowerPC embedded microprocessors that are used for automotive applications, such as engine control systems.

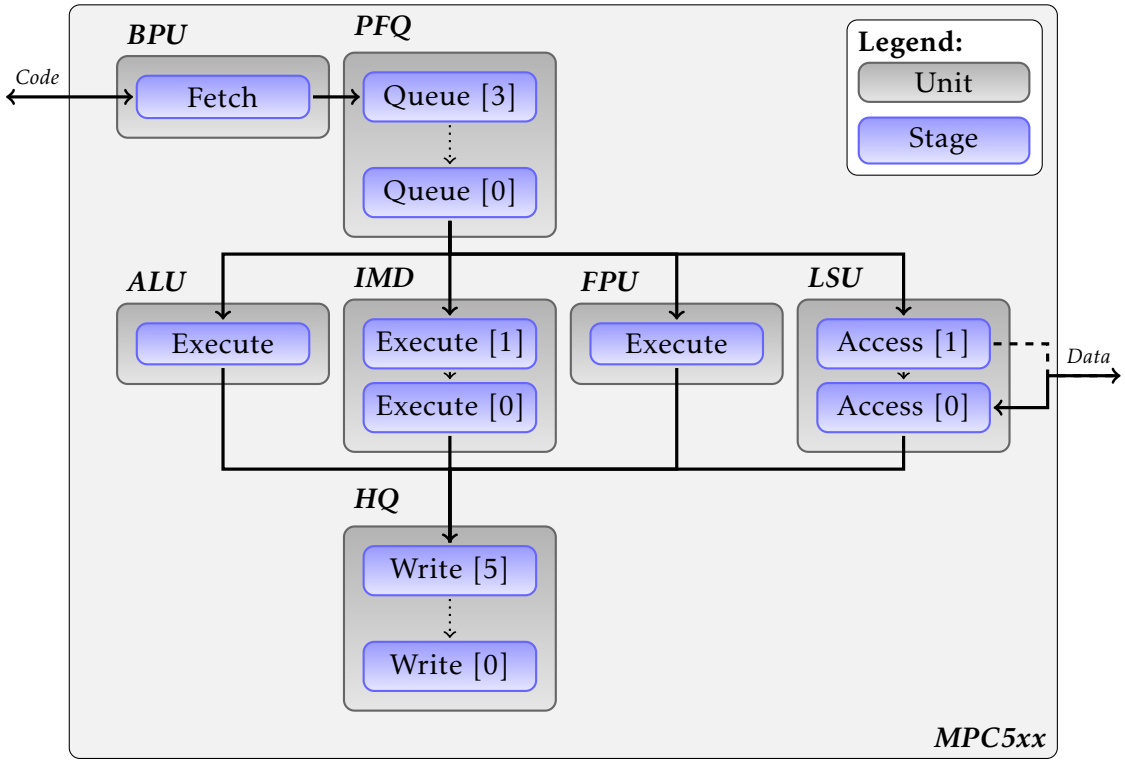



Figure 6.4: *MPC5xx Pipeline Model:* Processor model of the MPC5xx hardware architecture. The processor internal memories are not depicted. The branch processing unit (BPU) fetches instructions putting them into the instruction prefetch queue (PFQ). The PFQ dispatches a single instruction onto the corresponding functional unit, i.e., the arithmetic-logic unit (ALU), the integer multiplication and division unit (IMD), the floating-point unit (FPU), or the load-store unit (LSU). The history queue (HQ) records the update of the processor-internal registers in program order.

#### Hardware Description

The MPC5xx is a single-issue, out-of-order architecture that follows a Harvard architecture design. It has on-chip pipelined instruction and data buses that may address internal memory modules concurrently. The processor includes five independent execution units that may execute out-of-order. The branch processing unit (BPU) allows for a high instruction throughput by means of static branch prediction.

Up to four instructions may be prefetched into the prefetch queue (PFQ). The 32 bit arithmetic-logic unit (ALU) performs all kinds of integer operations except for multiplication and division, which are executed by a dedicated integer multiplier divider unit (IMD). The fully IEEE 754-compliant floating-point unit (FPU) is able to compute both single- and double-precision operations. Figure 6.4 depicts the manually designed pipeline model.

The MPC5xx processor family features a moderate amount of internal memory. Up to 1 MB of FLASH memory are available for code and data storage. The internal SRAM provides up to 36 KB for stack or scratch pad usage. A dedicated instruction-to-data bus module allows for code accesses on data bus and vice versa. An external bus interface shared between code and data bus connects up to 4 GB of external memory. The MPC5xx supports 8 bit, 16 bit and 32 bit external memories. To speed up program execution the processor may burst-fetch instructions from memory if the target memory supports burst operations.

Newer derivatives of the processor family (i.e., all MPC56x processors) feature a so-called branch-target buffer (BTB). The branch-target buffer is actually a small branch-target instruction cache (BTIC) using the FIFO replacement policy. The BTB may store up to four branch-target instructions for eight different branch targets. Analysis support for the BTB has been recently added to the static WCET analyzer aiT for MPC5xx [15].

### Measurement Setup

For our experiments we used an MPC565 evaluation board running at a clock frequency of 20 MHz. The processor features 1 MB internal FLASH memory, 32 KB internal SRAM, and 2 MB external SRAM with 32 bit port width.

Unlike the previously discussed processors, the MPC565 does not provide a full pin-out of the internal memory bus. Solely the external memory bus can be fully accessed by means of a logic analyzer. However, the evaluation board grants access to the instruction queue status pins `VF[2:0]`. The status pins denote the type of the last fetched instruction and indicate how many instructions have been flushed from the prefetch queue. This information allows for an in-depth view in the dispatch behavior of the processor. In this fashion we can achieve a per-action level trace validation by recording the following processor signals.

**CLKOUT** This output line is the external bus clock frequency, which can be configured to one-half of the internal system clock.

**ADDR[23:0]** Specifies the physical address of external bus transactions. The address is driven onto the bus and kept valid until a transfer acknowledge is received.

**BURST** Burst indicator that indicates whether the current transaction is a burst transaction or not.

**BDIP** Indicates that a burst access is ongoing.

**TS** The transfer start (TS) signal indicates the start of a bus cycle that transfers data to/from a slave device.

**TA** The transfer acknowledge (TA) signal indicates that the slave device addressed in the current transaction has accepted the data transferred by the master (write) or has driven the data bus with valid data (read). The slave device negates the TA signal after the end of the transaction.

**OE** The output enable (OE) signal that is asserted when a read access to an external slave is initiated.

**CS[3:0]** These output signals enable peripheral or memory devices at programmed addresses if defined appropriately in the memory controller.

**TSIZ[1:0]** Indicates the size of the requested data transfer in the current bus cycle.

**RD/WR** Indicates the direction of the data transfer for a transaction. A logic one indicates a read from a slave device, a logic zero indicates a write to a slave device.

**WE[3:0]** This output line is asserted when a write access to an external slave controlled by the memory controller is initiated by the chip. An active WE[*n*] bit indicates that the *n*-th byte of the data bus contains valid data to be stored.

**VF[2:0]** Visible instruction queue flush status. The VF signals indicate the type of the last fetched instruction (e.g., direct or indirect branch) and report the number of instructions flushed from the instruction queue in the internal core.

#### Results

We successfully performed a trace validation for an avionics application running on an MPC565 processor. Similar to the avionics application presented in Section 6.1.3, the application is implemented in a triple modular redundant fashion. By means of a shared bus the application instances match their (intermediate) results against each other.

The application successfully passes the trace validation. In total over 3.8 million lines of trace comprising about 370000 Nexus events (see Section 5.2) were automatically compared against the aiT MPC5xx pipeline model. Table 6.7 depicts an excerpt of the trace validation results. Due to NDA agreements we must not provide the real task names.

Even though the avionics application is data-dependent and thus most program paths cannot be excluded, the static WCET analysis performs quite well. The only additional sources for imprecision are the variable execution time of the integer division instruction and the variable latency for accesses on the shared bus. Hence we observe a difference between measurement and prediction of at most 10.63% and of 4.25% on average.

Table 6.8 shows the prediction graph size for each of the selected tasks. We can observe that the prediction graph complexity is not directly correlated to the distance between measurement and predicted bound. A prominent example is the task $\text{avionic}_5$ where the prediction graph comprises 10 times more events than observed whereas the difference between worst-case prediction and measurement amounts to about 3% only. The program control-flow allows for several paths, each of which is more or less equally long in terms of execution time.

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| $avionic_1$ | 10198 | 10800 | 5.90 |
| $avionic_2$ | 1473 | 1518 | 3.05 |
| $avionic_3$ | 1098 | 1198 | 9.11 |
| $avionic_4$ | 151 | 161 | 6.62 |
| $avionic_5$ | 4894 | 5040 | 2.98 |
| $avionic_6$ | 274 | 287 | 4.74 |
| $avionic_7$ | 1305 | 1358 | 4.06 |
| $avionic_8$ | 146 | 155 | 6.16 |
| $avionic_9$ | 3248 | 3333 | 2.62 |
| $avionic_{10}$ | 112 | 119 | 6.25 |
| $avionic_{11}$ | 1473 | 1517 | 2.99 |
| $avionic_{12}$ | 218 | 219 | 0.46 |
| $avionic_{13}$ | 207 | 209 | 0.97 |
| $avionic_{14}$ | 980 | 1021 | 4.18 |
| $avionic_{15}$ | 1473 | 1517 | 2.99 |
| $avionic_{16}$ | 10108 | 11182 | 10.63 |
| $avionic_{17}$ | 1282 | 1359 | 6.01 |
| $avionic_{18}$ | 1988 | 1996 | 0.40 |
| $avionic_{19}$ | 5471 | 5597 | 2.30 |
| $avionic_{20}$ | 241 | 254 | 5.39 |
| $avionic_{21}$ | 756 | 803 | 6.22 |
| $avionic_{22}$ | 274 | 289 | 5.47 |
| $avionic_{23}$ | 5084 | 5167 | 1.63 |
| $avionic_{24}$ | 986 | 1020 | 3.45 |
| $avionic_{25}$ | 1478 | 1518 | 2.71 |
| $avionic_{26}$ | 1288 | 1359 | 5.51 |
| $avionic_{27}$ | 5762 | 5942 | 3.12 |
| $avionic_{28}$ | 756 | 805 | 6.48 |
| $avionic_{29}$ | 274 | 288 | 5.11 |
| $avionic_{30}$ | 5090 | 5168 | 1.53 |

Table 6.7: *MPC5xx Analysis Results:* Trace validation for an avionics application running on an MPC565 processor. The difference between measured and predicted execution time is rather small. At worst we record a difference of 10.63%, on average of about 4.25%.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| $avionic_1$ | 334 | 20930 | 21293 | 886 | (x2.65) |
| $avionic_2$ | 69 | 6683 | 7302 | 349 | (x5.70) |
| $avionic_3$ | 58 | 2390 | 2477 | 227 | (x4.05) |
| $avionic_4$ | 18 | 640 | 701 | 83 | (x4.61) |
| $avionic_5$ | 252 | 28898 | 30766 | 2617 | (x10.38) |
| $avionic_6$ | 27 | 984 | 1051 | 107 | (x3.69) |
| $avionic_7$ | 84 | 6591 | 6986 | 661 | (x7.87) |
| $avionic_8$ | 11 | 907 | 948 | 104 | (x9.45) |
| $avionic_9$ | 142 | 11466 | 11899 | 803 | (x5.65) |
| $avionic_{10}$ | 13 | 779 | 864 | 102 | (x7.84) |
| $avionic_{11}$ | 69 | 4769 | 5143 | 405 | (x5.87) |
| $avionic_{12}$ | 21 | 283 | 306 | 42 | (x2.00) |
| $avionic_{13}$ | 7 | 1447 | 1478 | 64 | (x9.14) |
| $avionic_{14}$ | 80 | 1649 | 1780 | 198 | (x2.48) |
| $avionic_{15}$ | 69 | 4729 | 5101 | 401 | (x5.81) |
| $avionic_{16}$ | 498 | 67656 | 75829 | 4420 | (x8.88) |
| $avionic_{17}$ | 85 | 1189 | 1224 | 110 | (x1.29) |
| $avionic_{18}$ | 99 | 1653 | 1718 | 182 | (x1.83) |
| $avionic_{19}$ | 300 | 18499 | 19458 | 1783 | (x5.94) |
| $avionic_{20}$ | 24 | 1397 | 1506 | 142 | (x5.92) |
| $avionic_{21}$ | 38 | 3185 | 3372 | 262 | (x6.89) |
| $avionic_{22}$ | 28 | 301 | 318 | 38 | (x1.36) |
| $avionic_{23}$ | 223 | 3441 | 3622 | 376 | (x1.69) |
| $avionic_{24}$ | 80 | 1128 | 1232 | 151 | (x1.86) |
| $avionic_{25}$ | 69 | 4681 | 5047 | 397 | (x5.75) |
| $avionic_{26}$ | 85 | 1099 | 1136 | 102 | (x1.20) |
| $avionic_{27}$ | 319 | 11681 | 12262 | 1028 | (x3.22) |
| $avionic_{28}$ | 38 | 3713 | 3964 | 338 | (x8.89) |
| $avionic_{29}$ | 28 | 1243 | 1296 | 144 | (x5.15) |
| $avionic_{30}$ | 222 | 4625 | 4838 | 490 | (x2.21) |

Table 6.8: *MPC5xx Analysis Complexity:* From the prediction graph complexity we cannot directly conclude the distance between measurement and prediction. The task $avionic_5$ offers multiple program paths, each of which is more or less equally expensive. Hence the observed difference between measured execution time and predicted bound is rather small (2.98%).

### 6.1.5 MPC55xx

The MPC55xx processor family is a follow-up design of the MPC5xx hardware architecture. Like the MPC5xx processors, the MPC55xx CPUs are designed for automotive applications. Several MPC55xx derivatives are available, each of which based on the e200 processor core. Figure 6.5 depicts the manually designed MPC55xx (e200z6) processor model used for static WCET analysis.

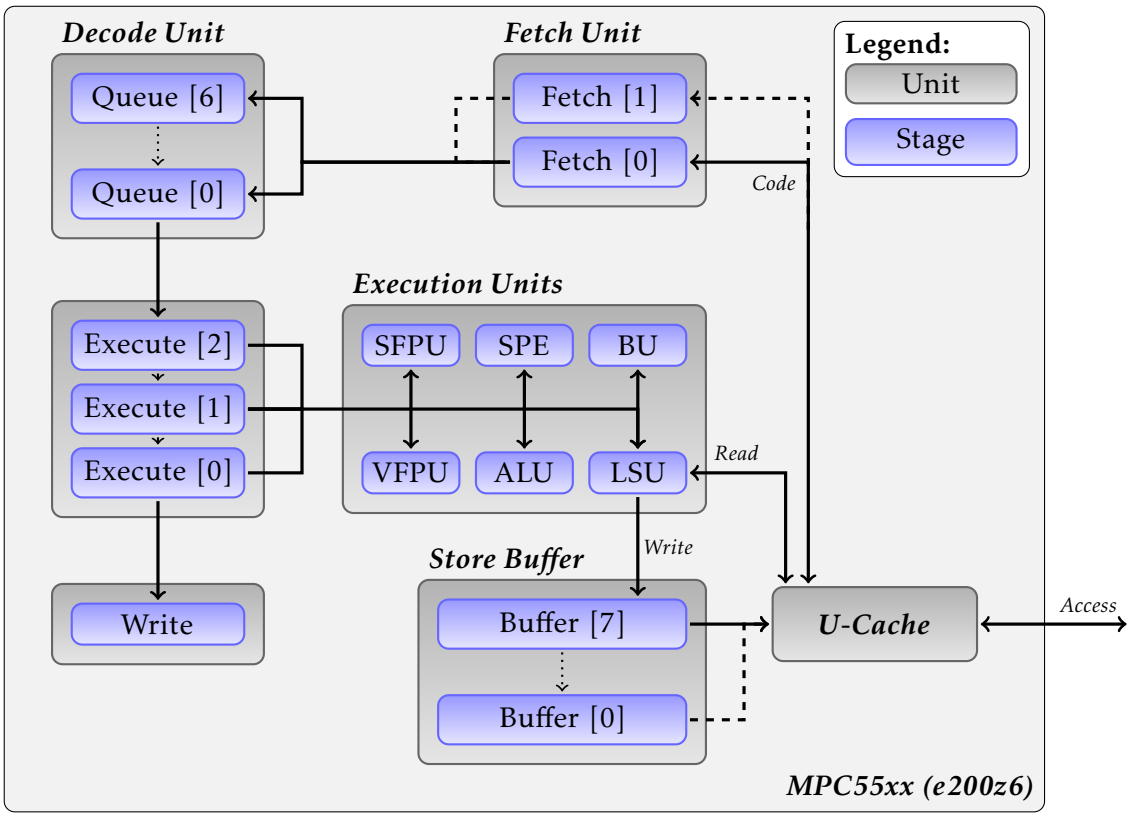



Figure 6.5: *MPC55xx Pipeline Model:* Processor model of an MPC55xx hardware architecture with e200z6 core. The unified cache delivers new instructions to the fetch unit, which can store up to two prefetched instructions. The fetch unit passes prefetched instructions to the decode unit that is able to hold seven instructions. The load-store unit (LSU) may defer write accesses to an eight-entry store buffer to reduce pipeline stalls.

#### Hardware Description

The MPC55xx is a single-issue, out-of-order von Neumann architecture.[11] Up to two instructions can be prefetched by the fetch unit before they are placed into

[11] Some MPC55xx derivatives are Harvard architectures, as e.g., the MPC5533.

the seven-entry decode unit. The decode unit may issue one instruction per cycle if no register dependencies between the current and unfinished instructions exist. Dispatched instructions can be allocated to six different execution units of which at most three can operate in parallel. The branch unit (BU) determines the branch target and informs the fetch unit if the program counter is to be adjusted. The arithmetic-logic unit (ALU) executes all kinds of integer operations. The scalar floating-point unit (SFPU) performs single-precision FPU computations. The vector floating-point unit (VFPU) provides SIMD floating-point computations on pairs of single-precision floats. The signal processing engine (SPE) provides SIMD integer arithmetic. The load-store unit (LSU) implements the memory interface. By means of an eight-entry store buffer the LSU can defer write accesses to reduce pipeline stalls.

MPC55xx processors feature an 32 KB unified (instruction and data) 8-way set-associative cache. Some derivatives even have a smaller unified cache with fewer lines per set. Entries are replaced using a pseudo-round-robin replacement algorithm using a set-global counter to determine the next line to be evicted. Both write-back and write-through cache write modes are supported. Unfortunately, the employed replacement policy makes a precise static analysis of the cache contents impossible.

The MPC55xx offers a large amount of internal memory that is connected via an internal crossbar with the processor core. Up to 3 MB of internal FLASH memory are available for code and data storage. The FLASH memory uses a FIFO-managed read buffer that stores up to two cache lines to speed up access times. The internal SRAM provides up to 192 KB for stack or scratch pad usage. An external bus interface (EBI) additionally adds up to 512 MB of external memory. The EBI supports 16 bit and 32 bit port sized external memories.

### Measurement Setup

For our experiments we used an MPC5566 evaluation board running at a clock frequency of up to 132 MHz. The processor features 2 MB internal FLASH memory, 64 KB internal SRAM, and 2 MB external SRAM with 32 bit port width.

The MPC5566 does not provide a full pin-out of the internal memory bus. Solely the external memory bus can be fully accessed by means of a logic analyzer. But the external bus is driven at half (or quarter) the processor clock speed. Hence, observing the external memory bus allows only for rather coarse conclusion about the pipeline behavior. By means of a logic analyzer we record the following signals.

**CLKOUT** This output line is the external bus clock frequency, which runs at half the processor clock speed. During our measurements the internal clock frequency was set up to 25 MHz. The external bus clock frequency was hence 12.5 MHz.

**ADDR[23:0]** Specifies the physical address of external bus transactions. The address is driven onto the bus and kept valid until a transfer acknowledge is received. The upper 8 bit of the 32 bit address are not made available.

**TS** The transfer start (TS) signal indicates the start of an external bus cycle that transfers data to/from a slave device.

**CS[3:0]** These output signals enable peripheral or memory devices at programmed addresses if defined appropriately in the memory controller.

**TSIZ[1:0]** Indicates the size of the requested data transfer in the current bus cycle.

**RD/WR** Indicates whether an external bus transfer is a read or write operation.

**BDIP** Indicates that an EBI burst transfer is in progress. A burst access can only be triggered by a cache miss. Accesses to uncached memory cannot provoke a burst operation.

**WE[3:0]** The write-enable bits specify which data pins contain valid data for an external bus transfer. An active `WE[`*n*`]` bit indicates that the *n*-th byte of the data bus contains valid data to be stored.

**TA** The transfer acknowledge (TA) signal is asserted by the EBI owner to acknowledge that the slave has completed the current transfer.

### Results

In total we have designed 225 test cases to investigate the pipeline behavior of the MPC55xx architecture. All tests successfully pass the trace validation. Table 6.9 shows an excerpt of the validation results for the MPC5566 test programs. On average we observe a gap between measurement and prediction of about 1.02% only. The worst difference to observe is 5.19% for the test `branch-predict`. This is caused by the variable execution time of the integer division instruction `divw`, which may complete between 6 and 16 processor cycles. The pipeline analysis is unable to statically determine when this instruction finishes. An interesting observation is that enabling the store buffer can lead to a worse performance for some programs. With enabled store buffer the processor takes 60 cycles longer to execute the same input program (see `store-buffer-enabled` and `store-buffer-disabled`).

Table 6.10 compares the number of measured events with the prediction graph size. We observe that the complexity is increasing for test cases related to the cache. This is expected due to the bad analyzability of the cache replacement policy, which prevents the static analysis to ever determine precise cache contents. The analysis complexity further increases if a memory access might hit the cache line that is currently being filled. In this case the state space of the analysis increases significantly which has a huge impact on the prediction graph (see `streaming-cache`).

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| alu-stall | 530 | 530 | 0.00 |
| branch | 1560 | 1578 | 1.15 |
| branch-predict | 2314 | 2434 | 5.19 |
| branch-spill | 862 | 862 | 0.00 |
| call | 458 | 478 | 4.37 |
| load-code-cache-hit$_1$ | 172 | 172 | 0.00 |
| load-code-cache-hit$_2$ | 118 | 122 | 3.39 |
| load-code-cache-miss | 124 | 126 | 1.61 |
| load-flash | 4300 | 4308 | 0.19 |
| load-multiple-stall | 410 | 415 | 1.22 |
| load-sram | 1792 | 1798 | 0.33 |
| load-store | 366 | 366 | 0.00 |
| load-store-cache$_1$ | 362 | 362 | 0.00 |
| load-store-cache$_2$ | 268 | 272 | 1.49 |
| load-store-cache$_3$ | 260 | 264 | 1.54 |
| loop$_1$ | 9718 | 9920 | 2.08 |
| loop$_2$ | 9738 | 9938 | 2.05 |
| loop$_3$ | 9274 | 9274 | 0.00 |
| store-buffer-enabled | 1102 | 1102 | 0.00 |
| store-buffer-disabled | 1042 | 1042 | 0.00 |
| store-load | 366 | 366 | 0.00 |
| store-load-cache | 162 | 162 | 0.00 |
| store-store | 366 | 366 | 0.00 |
| streaming-cache | 208 | 211 | 1.44 |
| vle-alu | 62 | 62 | 0.00 |
| vle-alu-mul | 126 | 128 | 1.59 |
| vle-branch | 264 | 268 | 1.52 |
| vle-branch-spill | 860 | 860 | 0.00 |
| vle-load | 98 | 98 | 0.00 |
| vle-load-multiple | 306 | 308 | 0.65 |
| vle-load-store | 236 | 238 | 0.85 |

Table 6.9: *MPC55xx Analysis Results:* Trace validation results for aiT for MPC55xx. Especially due to the bad analyzability of the unified cache the analysis is unable to precisely predict the measured execution behavior. At worst, we recognize a difference to the highest measured execution time of 5.19%.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| alu-stall | 204 | 246 | 246 | 216 | (x1.06) |
| branch | 596 | 964 | 975 | 784 | (x1.32) |
| branch-predict | 876 | 2636 | 2927 | 1861 | (x2.12) |
| branch-spill | 336 | 362 | 362 | 348 | (x1.04) |
| call | 160 | 248 | 251 | 172 | (x1.07) |
| load-code-cache-hit$_1$ | 66 | 702 | 871 | 445 | (x6.74) |
| load-code-cache-hit$_2$ | 50 | 308 | 350 | 197 | (x3.94) |
| load-code-cache-miss | 52 | 391 | 455 | 263 | (x5.06) |
| load-flash | 1010 | 1357 | 1358 | 1022 | (x1.01) |
| load-multiple-stall | 152 | 1003 | 1071 | 949 | (x6.24) |
| load-sram | 650 | 912 | 913 | 662 | (x1.02) |
| load-store | 136 | 165 | 165 | 148 | (x1.09) |
| load-store-cache$_1$ | 134 | 1472 | 1862 | 1082 | (x8.07) |
| load-store-cache$_2$ | 99 | 356 | 427 | 275 | (x2.78) |
| load-store-cache$_3$ | 97 | 315 | 355 | 234 | (x2.41) |
| loop$_1$ | 3444 | 4673 | 4673 | 3456 | (x1.00) |
| loop$_2$ | 3452 | 4682 | 4682 | 3464 | (x1.00) |
| loop$_3$ | 3227 | 4551 | 4551 | 3239 | (x1.00) |
| store-buffer-enabled | 388 | 505 | 505 | 396 | (x1.02) |
| store-buffer-disabled | 388 | 489 | 489 | 400 | (x1.03) |
| store-load | 136 | 165 | 165 | 148 | (x1.09) |
| store-load-cache | 64 | 642 | 806 | 453 | (x7.08) |
| store-store | 136 | 165 | 165 | 148 | (x1.09) |
| streaming-cache | 84 | 5406 | 6843 | 3931 | (x46.80) |
| vle-alu | 24 | 265 | 319 | 138 | (x5.75) |
| vle-alu-mul | 47 | 400 | 491 | 234 | (x4.98) |
| vle-branch | 103 | 236 | 268 | 175 | (x1.70) |
| vle-branch-spill | 335 | 551 | 596 | 451 | (x1.35) |
| vle-load | 37 | 101 | 109 | 69 | (x1.86) |
| vle-load-multiple | 117 | 551 | 568 | 504 | (x4.31) |
| vle-load-store | 83 | 138 | 140 | 123 | (x1.48) |

Table 6.10: *MPC55xx Analysis Complexity:* Number of measured events and corresponding prediction graph size. The prediction graph size increases with a higher variability of the processor's execution behavior. This mostly applies to tests related to the unified cache.

### 6.1.6 Intel 386

The Intel 386 was the first 32 bit x86 microprocessor that was employed in many workstations after its release. Nowadays embedded versions of the original Intel 386 design are used in aerospace technology. Figure 6.6 depicts the manually designed processor model used for static WCET analysis.

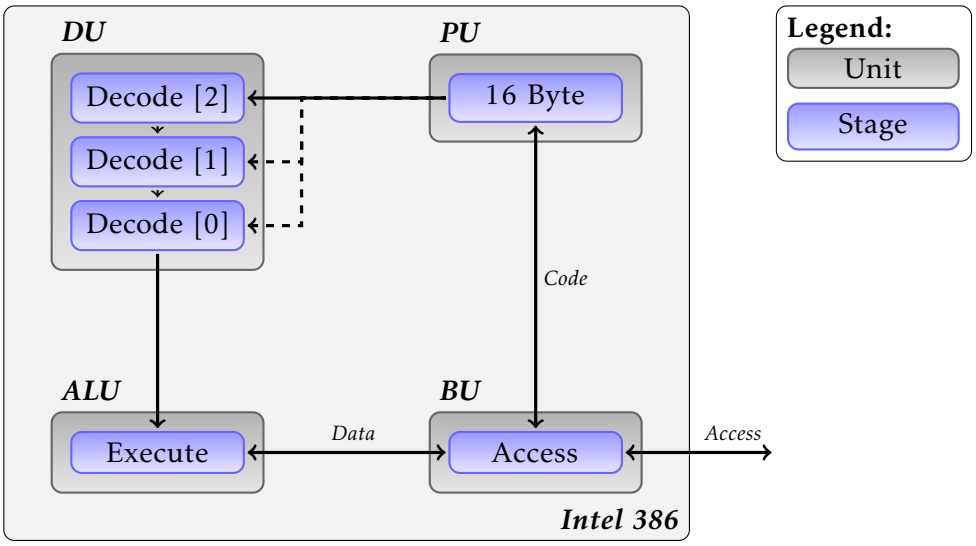



Figure 6.6: *Intel 386 Pipeline Model:* Processor model for the Intel 386 hardware architecture. The prefetch unit (PU) stores up to 16 bytes of raw code and transfers decoded instruction segments into the decode unit (DU). Up to three instructions can take place in the DU before the prefetching is stalled. The arithmetic-logic unit (ALU) executes one instruction after another and initiates memory accesses. The bus unit (BU) provides a 32 bit interface to main memory.

#### Hardware Description

The Intel 386 is a single-issue, in-order von Neumann hardware architecture. Up to 16 bytes of raw code can be prefetched into the prefetch unit (PU) before the decoded instruction segments are placed into a three-entry decode unit (DU). The arithmetic-logic unit (ALU) can execute one instruction at a time. Floating-point instructions are not supported (FPU support requires a math co-processor, such as the Intel 80387).

Via dynamic bus sizing the processor is able to directly communicate with 16 bit or 32 bit memories. The flat memory model of the Intel 386 allows a continuous addressing of the 32 bit address space. Earlier x86 processors supported only very small linearly addressable memory regions.

### Measurement Setup

The Intel 386 allows for a trace validation on a per-action level. For comparison with the pipeline model we record a bus trace by means of the following signals.

**CLK2** CLK2 provides the fundamental timing for the Intel 386. It is internally divided by two to generate the internal processor clock used for instruction execution. The processor is clocked at a frequency of 16 MHz.

**ADDR[31:2]** The address bus provides physical memory addresses or I/O port addresses. The bus unit implicitly splits up a misaligned access into separate aligned accesses.

**ADS** The address status signal (ADS) indicates that a valid address is being latched at the address bus.

**NA** The next address request signal (NA) is asserted to request address pipelining. It indicates that the processor is able to accept new data on the data bus.

**BS16** The bus size signal (BS16) allows the processor to directly access 16 bit or 32 bit data buses.

**TA** The transfer-acknowledge signal (TA) denotes that the current bus cycle is complete.

### Results

For the trace validation we have had access to measurements from an avionics application executed on an Intel 386 embedded processor. The system is clocked at 16 MHz and comprises 2 MB of on-board PROM and 256 KB internal SRAM memory.

The application successfully passes the trace validation. In total over 21000 bus events were recorded and automatically compared with the aiT Intel 386 pipeline model. Table 6.11 depicts an excerpt of the trace validation. On average we observe a difference between measured and prediction execution time of about 1.36%. Table 6.12 displays the size of the prediction graph compared to measured events. Here, the prediction graph complexity is mainly influenced by the program control-flow.

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| $avionic_1$ | 1114 | 1121 | 0.63 |
| $avionic_2$ | 1553 | 1579 | 1.67 |
| $avionic_3$ | 444 | 448 | 0.90 |
| $avionic_4$ | 406 | 406 | 0.00 |
| $avionic_5$ | 2919 | 3059 | 4.80 |
| $avionic_6$ | 3487 | 3509 | 0.63 |
| $avionic_7$ | 981 | 997 | 1.63 |
| $avionic_8$ | 367 | 376 | 2.45 |
| $avionic_9$ | 1065 | 1065 | 0.00 |
| $avionic_{10}$ | 769 | 769 | 0.00 |
| $avionic_{11}$ | 515 | 526 | 2.14 |
| $avionic_{12}$ | 641 | 654 | 2.03 |
| $avionic_{13}$ | 449 | 453 | 0.89 |
| $avionic_{14}$ | 490 | 490 | 0.00 |
| $avionic_{15}$ | 293 | 302 | 3.07 |
| $avionic_{16}$ | 4098 | 4154 | 1.37 |
| $avionic_{17}$ | 1057 | 1069 | 1.14 |
| $avionic_{18}$ | 854 | 854 | 0.00 |
| $avionic_{19}$ | 222 | 222 | 0.00 |
| $avionic_{20}$ | 383 | 386 | 0.78 |
| $avionic_{21}$ | 168 | 176 | 4.76 |
| $avionic_{22}$ | 665 | 670 | 0.75 |
| $avionic_{23}$ | 464 | 466 | 0.43 |
| $avionic_{24}$ | 830 | 840 | 1.20 |
| $avionic_{25}$ | 1531 | 1531 | 0.00 |
| $avionic_{26}$ | 960 | 971 | 1.15 |
| $avionic_{27}$ | 1444 | 1474 | 2.08 |
| $avionic_{28}$ | 1019 | 1026 | 0.69 |
| $avionic_{29}$ | 1111 | 1133 | 1.98 |
| $avionic_{30}$ | 1111 | 1151 | 3.60 |

Table 6.11: *Intel 386 Analysis Results:* Trace validation for an avionics application running on an Intel 386 processor. The difference between measured and predicted execution time is rather small. We observe a difference of 4.8% at worst and of 1.36% on average.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| $avionic_1$ | 244 | 1808 | 1952 | 389 | (x1.59) |
| $avionic_2$ | 358 | 4447 | 5010 | 1520 | (x4.25) |
| $avionic_3$ | 136 | 3238 | 3751 | 1379 | (x10.14) |
| $avionic_4$ | 114 | 1289 | 1447 | 537 | (x4.71) |
| $avionic_5$ | 524 | 13209 | 15113 | 3085 | (x5.89) |
| $avionic_6$ | 852 | 6350 | 7127 | 2134 | (x2.50) |
| $avionic_7$ | 260 | 1273 | 1334 | 561 | (x2.16) |
| $avionic_8$ | 96 | 1382 | 1559 | 627 | (x6.53) |
| $avionic_9$ | 250 | 2409 | 2675 | 921 | (x3.68) |
| $avionic_{10}$ | 182 | 2179 | 2414 | 770 | (x4.23) |
| $avionic_{11}$ | 126 | 3526 | 4042 | 1125 | (x8.93) |
| $avionic_{12}$ | 162 | 2234 | 2462 | 772 | (x4.77) |
| $avionic_{13}$ | 128 | 1300 | 1398 | 594 | (x4.64) |
| $avionic_{14}$ | 152 | 947 | 1049 | 450 | (x2.96) |
| $avionic_{15}$ | 82 | 1327 | 1504 | 613 | (x7.48) |
| $avionic_{16}$ | 900 | 9377 | 10344 | 3291 | (x3.66) |
| $avionic_{17}$ | 242 | 2895 | 3228 | 1072 | (x4.43) |
| $avionic_{18}$ | 196 | 2262 | 2502 | 789 | (x4.03) |
| $avionic_{19}$ | 46 | 342 | 366 | 112 | (x2.43) |
| $avionic_{20}$ | 54 | 565 | 620 | 138 | (x2.56) |
| $avionic_{21}$ | 64 | 238 | 257 | 131 | (x2.05) |
| $avionic_{22}$ | 210 | 1787 | 1957 | 817 | (x3.89) |
| $avionic_{23}$ | 136 | 1394 | 1522 | 636 | (x4.68) |
| $avionic_{24}$ | 198 | 2153 | 2347 | 794 | (x4.01) |
| $avionic_{25}$ | 344 | 2465 | 2687 | 603 | (x1.75) |
| $avionic_{26}$ | 218 | 1770 | 1885 | 482 | (x2.21) |
| $avionic_{27}$ | 340 | 2016 | 2169 | 382 | (x1.12) |
| $avionic_{28}$ | 262 | 900 | 911 | 293 | (x1.12) |
| $avionic_{29}$ | 250 | 1992 | 2159 | 362 | (x1.45) |
| $avionic_{30}$ | 204 | 4682 | 5422 | 1104 | (x5.41) |

Table 6.12: *Intel 386 Analysis Complexity:* Number of measured events and corresponding prediction graph size. The prediction graph size increases with a higher variability of the processor's execution behavior. Here the program control-flow contributes the most to the prediction graph size.

### 6.1.7 AMD 486

The AMD 486 processor is an 80486 derivative, which is a successor of the 80386 hardware architecture. A higher performance is achieved by adding a unified instruction and data cache and an integrated floating-point unit to the processor. Figure 6.7 depicts the manually designed processor model used for static WCET analysis.

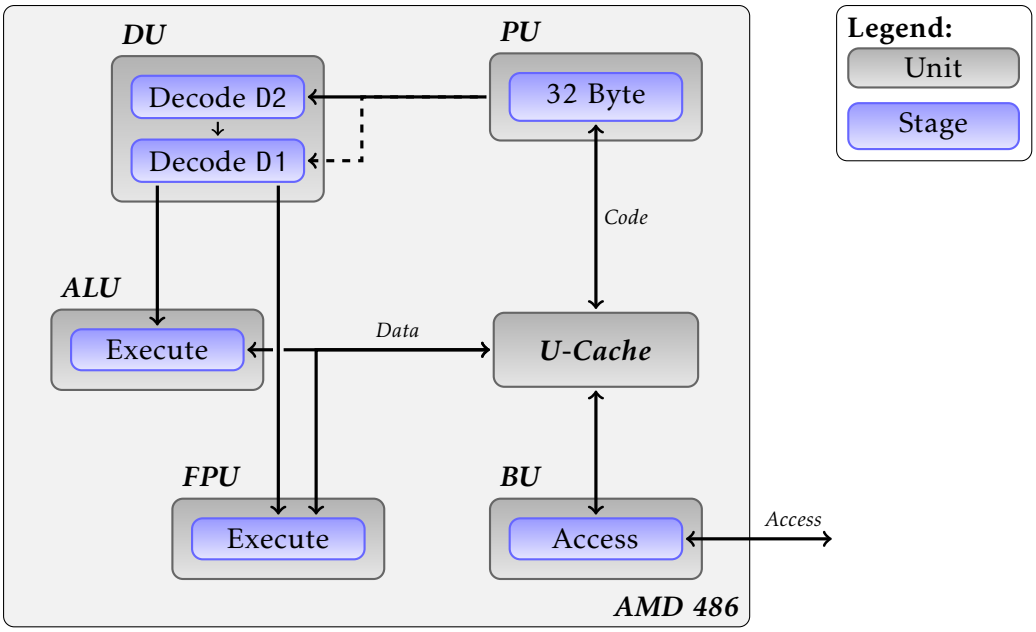



Figure 6.7: *AMD 486 Pipeline Model:* Processor model for the AMD 486 hardware architecture. The prefetch unit (PU) stores up to 32 bytes of raw code and transfers decoded instruction segments into the decode unit (DU). The arithmetic-logic unit (ALU) and the floating-point unit (FPU) can operate in parallel. The bus unit (BU) provides a 32 bit interface to main memory.

#### Hardware Description

The AMD 486 is a single-issue, out-of-order von Neumann hardware architecture. The prefetch unit (PU) transfers the raw instruction stream from the on-chip cache or from main memory via the bus unit (BU). Up to 32 bytes of code can be prefetched. The decode unit (DU) decodes the code stream in two stages `D1` and `D2` into microcode that is executed on either the arithmetic-logic unit (ALU) or the floating-point unit (FPU). Parallelism between the ALU and the FPU exists but is limited because both units share the memory interface.

### Measurement Setup

As the Intel 386, the AMD 486 allows for a trace validation on a per-action level. For comparison with the pipeline mode we record a bus trace by means of the following signals.

**CLK** The CLK input provides the basic microprocessor timing signal. An additional multiplier value is used to generate the internal operating frequency for AMD 486 processor.

**ADDR[31:2]** The address bus provides physical memory or I/O port addresses.

**ADS** The address status signal (ADS) indicates that a valid address is being latched at the address bus.

**BS8/BS16** The bus size signals (BS8, BS16) allow the processor to directly access 8 bit, 16 bit or 32 bit data buses.

**W/R** The write/read output signal indicates the current access type.

**RDY** The non-burst ready signal (RDY) indicates that the current bus cycle is complete and valid data is present on the data bus.

**BRDY** The burst ready signal (BRDY) behaves similar to RDY during a burst fetch bus operation.

### Results

For the trace validation we had access to measurements from an avionics application executed on an AMD 486 DX4 processor clocked at 96 MHz. The DX4 has a 4-way set-associative 16 KB unified cache that uses the write-through cache write policy. Cache entries are replaced via a pseudo-least-recently-used (PLRU) replacement strategy. In addition to the cache, the processor memories comprise about 4 MB on-board PROM and 512 KB SRAM.

The application successfully passes the trace validation. Around 30000 bus events are available for an automatic validation of the aiT AMD 486 processor pipeline model. Table 6.13 depicts an excerpt of the trace validation. Compared to the Intel 386, the static analysis for the AMD 486 performs a bit worse. This is caused by the unified cache and the variable execution time of floating-point instructions. On average we observe a difference of about 3.74%.

Table 6.14 displays the size of the prediction graph compared to measured events. As expected the complexity of the prediction graph is generally higher than for the Intel 386 WCET analysis. The main reason for this behavior is the statically unpredictable timing behavior of the FPU. For example, a floating-point sine operation may take between 67 and 395 processor cycles, depending on the operand values.

| Test | Measurement | Prediction | |
|---|---|---|---|
| | (cycles) | Bound (cycles) | Distance (%) |
| $avionic_1$ | 1509 | 1580 | 4.71 |
| $avionic_2$ | 5454 | 5685 | 4.24 |
| $avionic_3$ | 5505 | 5872 | 6.67 |
| $avionic_4$ | 6654 | 6904 | 3.76 |
| $avionic_5$ | 2277 | 2343 | 2.90 |
| $avionic_6$ | 4866 | 4891 | 0.51 |
| $avionic_7$ | 7413 | 7452 | 0.53 |
| $avionic_8$ | 1911 | 1956 | 2.35 |
| $avionic_9$ | 1305 | 1341 | 2.76 |
| $avionic_{10}$ | 1287 | 1390 | 8.00 |
| $avionic_{11}$ | 2334 | 2517 | 7.84 |
| $avionic_{12}$ | 2487 | 2688 | 8.08 |
| $avionic_{13}$ | 1911 | 1965 | 2.83 |
| $avionic_{14}$ | 1023 | 1053 | 2.93 |
| $avionic_{15}$ | 2817 | 2898 | 2.88 |
| $avionic_{16}$ | 2634 | 2718 | 3.19 |
| $avionic_{17}$ | 2610 | 2700 | 3.45 |
| $avionic_{18}$ | 3537 | 3627 | 2.54 |
| $avionic_{19}$ | 3513 | 3618 | 2.99 |
| $avionic_{20}$ | 1998 | 2094 | 4.80 |
| $avionic_{21}$ | 1980 | 2070 | 4.55 |
| $avionic_{22}$ | 1950 | 1989 | 2.00 |
| $avionic_{23}$ | 1380 | 1434 | 3.91 |
| $avionic_{24}$ | 2886 | 2976 | 3.12 |
| $avionic_{25}$ | 1365 | 1374 | 0.66 |
| $avionic_{26}$ | 1194 | 1243 | 4.10 |
| $avionic_{27}$ | 1818 | 1879 | 3.36 |
| $avionic_{28}$ | 1566 | 1644 | 4.98 |
| $avionic_{29}$ | 1815 | 1879 | 3.53 |
| $avionic_{30}$ | 2220 | 2317 | 4.37 |

Table 6.13: *AMD 486 Analysis Results:* Trace validation for an avionics application running on an AMD 486 processor. The observed difference between measurements and prediction is caused by the variable execution time of floating-point instructions. At worst we record a difference of 8.08%, on average of 3.74%.

| Test | Measurement | Prediction | | | |
|---|---|---|---|---|---|
| | Events | Nodes | Edges | Events | (Factor) |
| $avionic_1$ | 375 | 3632 | 4563 | 3214 | (x8.57) |
| $avionic_2$ | 1287 | 13356 | 15740 | 13354 | (x10.38) |
| $avionic_3$ | 1326 | 17810 | 21101 | 17539 | (x13.23) |
| $avionic_4$ | 1624 | 12999 | 14919 | 13067 | (x8.05) |
| $avionic_5$ | 577 | 3157 | 3523 | 3411 | (x5.91) |
| $avionic_6$ | 969 | 9994 | 11102 | 9257 | (x9.55) |
| $avionic_7$ | 1225 | 12963 | 14821 | 9872 | (x8.06) |
| $avionic_8$ | 498 | 2265 | 2382 | 2635 | (x5.29) |
| $avionic_9$ | 357 | 659 | 685 | 763 | (x2.14) |
| $avionic_{10}$ | 217 | 676 | 791 | 374 | (x1.72) |
| $avionic_{11}$ | 524 | 1548 | 1595 | 1342 | (x2.56) |
| $avionic_{12}$ | 563 | 1607 | 1658 | 1396 | (x2.48) |
| $avionic_{13}$ | 417 | 2051 | 2213 | 2209 | (x5.30) |
| $avionic_{14}$ | 251 | 1270 | 1336 | 1394 | (x5.55) |
| $avionic_{15}$ | 709 | 2118 | 2236 | 2332 | (x3.29) |
| $avionic_{16}$ | 673 | 2118 | 2236 | 2332 | (x3.47) |
| $avionic_{17}$ | 675 | 2118 | 2236 | 2332 | (x3.45) |
| $avionic_{18}$ | 896 | 2951 | 3138 | 3217 | (x3.59) |
| $avionic_{19}$ | 888 | 2665 | 2811 | 2893 | (x3.26) |
| $avionic_{20}$ | 546 | 1195 | 1246 | 1313 | (x2.40) |
| $avionic_{21}$ | 545 | 1195 | 1246 | 1313 | (x2.41) |
| $avionic_{22}$ | 506 | 1476 | 1552 | 1604 | (x3.17) |
| $avionic_{23}$ | 378 | 737 | 765 | 797 | (x2.11) |
| $avionic_{24}$ | 780 | 1706 | 1774 | 1895 | (x2.43) |
| $avionic_{25}$ | 373 | 970 | 1018 | 1063 | (x2.85) |
| $avionic_{26}$ | 227 | 1101 | 1181 | 772 | (x3.40) |
| $avionic_{27}$ | 405 | 1588 | 1689 | 1785 | (x4.41) |
| $avionic_{28}$ | 327 | 783 | 811 | 886 | (x2.71) |
| $avionic_{29}$ | 405 | 1587 | 1688 | 1785 | (x4.41) |
| $avionic_{30}$ | 460 | 2054 | 2165 | 1950 | (x4.24) |

Table 6.14: *AMD 486 Analysis Complexity:* Number of measured events and corresponding prediction graph size. The variable execution time of floating-point instructions together with the data-dependent control flow impact the complexity of the prediction graph.

## 6.2 Timing Anomalies

By means of the automatic timing anomaly detection method introduced in Chapter 4 we examined the presence of timing anomalies for the hardware architectures subject to trace validation in Section 6.1. We carefully investigated each found instance of timing anomalies and attempted to identify their cause. Due to the inevitable loss of information caused by abstract interpretation, the static analysis might trigger an execution behavior that cannot occur in reality. Section 6.2.5 discusses one such *virtual* timing anomaly (see Definition 4.1 on page 51) that is caused by the meet-operator for abstract cache states.

### 6.2.1 ERC32

For the SPARC v7 ERC32 hardware architecture, we automatically processed over 1000 programs. None of the test programs provided evidence for the presence of a timing anomaly.

This result is not unexpected. Due to the lack of an instruction or data cache, cache-related timing anomalies simply cannot occur in the ERC32 hardware architecture. Furthermore, we did not expect to observe any kind of speculation anomaly because the processor is always stalled upon a resource conflict. Even though one floating-point instruction can be executed in parallel to non-FPU instructions, the pipeline progression immediately stops if another FPU instruction is about to be dispatched that conflicts with the currently executed one. No other activity other than waiting for the conflict to resolve takes place in the ERC32 processor in that situation. Due to this behavior, we also did not expect for scheduling anomalies to occur. The ERC32 processor is thus considered to be a fully timing compositional hardware architecture.

### 6.2.2 LEON2

The SPARC v8 LEON2 processor suffers from a timing anomaly that is related to the implementation of the instruction cache and the cache line fill mechanism. Each cache line comprises a valid bit per word, which allows for partially filled cache lines. Upon a cache miss the processor initiates a burst code-fetch operation to fill the cache line starting from the requested instruction word until the last instruction in the cache line. The processor does not perform wrap-around (or cyclic) burst fetches. Hence, cache lines are not always filled completely. For example, if the initial code request hits the second word of a cache line, the processor will fill up the cache line starting from the second word to the last word of that cache line. In this case the first instruction word will not be fetched and placed into the cache line. In addition to this code fetch behavior, the cache line fill operation can be interrupted.

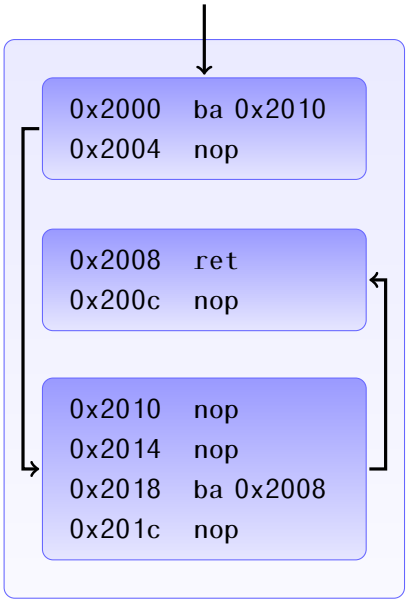



Figure 6.8: *Code Snippet triggering LEON2 Timing Anomaly:* Minimal code snippet that reveals a timing anomaly. The cache line filling mechanism allows for partially filled cache lines. An initial cache hit for the first instruction at 0x2000 leads to additional instruction fetches that would otherwise not occur.

After assuming an initial cache hit the valid bits of the remaining instructions in the corresponding cache line can thus accept any value. Due to this hardware mode of behavior we can observe a *speculation timing anomaly*.

The code sequence in Figure 6.8 triggers this timing anomaly as depicted in Figure 4.4 on page 61. Unfortunately, we were not able to force the abortion of a cache line fill operation and could thus not measure this behavior in the real device. In principle this would require triggering an interrupt at the right time, which is infeasible in practice.

In addition to this anomaly, we are also able to show the presence of another speculation timing anomaly that is related to the processor's ability to serve code fetch requests while a cache line fill operation is going on (i.e., *hit under fill*). The occurrence of this timing anomaly also depends on partially filled cache lines, but does not require the cache line fill operation to be aborted to trigger this anomaly.

An initial cache miss to 0x2000 causes the memory controller to fill the whole cache line. One after another the controller fetches the instructions from main memory and fast-forwards them to the integer pipeline. Independent of the state of valid bits of the subsequent words inside the cache line, the memory controller will refetch the

whole cache line. The processor pipeline is unaware of the cache line fill operation, and issues the next code request after receiving the first instruction. Since we cannot be sure about the status of the valid bits, the new code request might hit or miss the cache. For a timing anomaly now to occur, the memory access timing plays an important role. Here we choose an access latency of zero wait states.

If the next code request would hit the cache, i.e., the corresponding valid bit is already true, it might happen that the requested word is just arriving from main memory. In that case the memory controller is unable to instantly serve the code fetch request because it first needs to update the cache line. One cycle after the cache line update, the memory controller delivers the requested instruction to the processor. But if the next code request would not hit the cache, the memory controller would be able to deliver the corresponding instruction in the very same cycle it arrives from main memory. Figure 6.9 depicts the automatically detected instance of this anomaly.

The above timing anomalies are both $k$-bounded timing anomalies. Their timing impact is limited by the size of a cache line, i.e., the number of valid bits per line. After visiting the whole cache line, the processor state eventually stabilizes because the processor pipeline stalls until the cache line fill operation is complete.

In total we automatically processed over 1000 programs in search for other timing anomalies than the ones discussed above. The set of programs comprises rather simple benchmark programs as well as real-world applications from the space domain. We could not find any other timing anomaly instance except for the speculation anomalies caused by the implementation of the instruction cache. As the LEON2 processor pipeline is quite similar to the ERC32 instruction pipeline, this observation was expected. Except for the instruction cache there is no other hardware component that could trigger a timing anomaly. Upon resource conflicts the processor stalls and waits until the conflict is being resolved. In accordance to our findings, the LEON2 is classified as compositional architecture with $k$-bounded effects.

### 6.2.3 M68020

We automatically processed about 2000 programs to investigate whether the M68020 processor suffers from timing anomalies. The test bench comprises benchmark programs and a few avionics applications.

None of the applications subject to investigation revealed a timing anomaly. This observation does not come unexpected due to the rather simple structure of the M68020/M68882 processor. Like the ERC32 (see Section 6.2.1), the processor always stalls upon a resource dependency (e.g., if waiting for memory, etc.). Even though

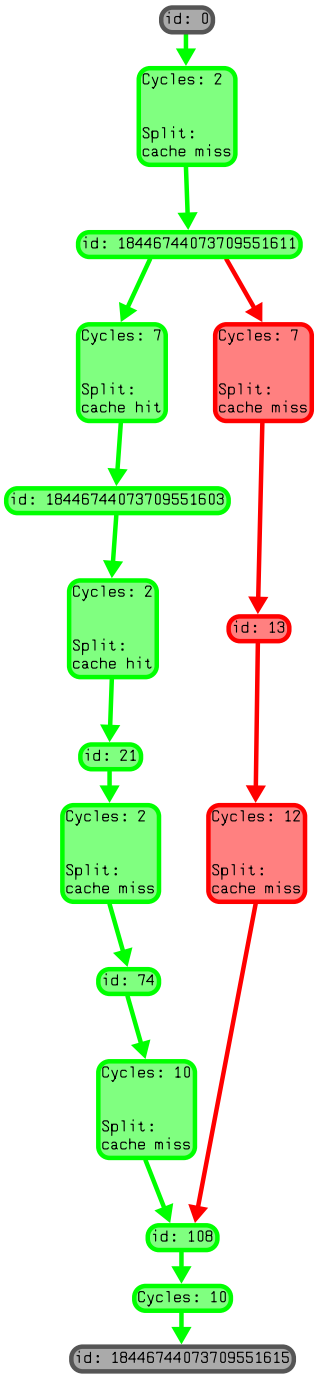



Figure 6.9: *LEON2 Hit-Under-Fill Speculation Anomaly:* Excerpt of the prediction graph for the analysis of the code snippet shown in Figure 6.8. On the worst-case path (green edges) there are two cache hit states that cause an additional 2-cycle penalty (hit under fill). The red edges denote the execution where only local worst-case behavior is being considered.

the M68882 floating-point co-processor is able to execute up to two FPU instructions in parallel, a faster execution of an FPU instruction does not induce some kind of penalty that would not occur otherwise. Either an FPU instruction is fully executed in parallel to subsequent instructions or the M68020 is stalled until the resource conflict is being resolved. Due to this behavior timing anomalies are unexpected. Hence, the M68020 processor is considered as fully timing compositional hardware architecture.

### 6.2.4 MPC5xx

We had over 1200 different test programs at our disposal to discover possible instances of timing anomalies in the MPC5xx hardware architecture. The set of test cases comprises simple benchmark programs, some automotive as well as some avionics applications. No analysis result of the available programs provided evidence about the presence of timing anomalies in the MPC5xx processor.

Despite the rather complex pipeline structure – as compared to ERC32, LEON2, or M68020 – we actually did not expect any timing anomalies. Keeping track of the outcome of up to six instructions, the processor is able to execute instructions out-of-order. The execution units present in the MPC5xx hardware architecture are dedicated to a certain kind of operation. As the execution units cannot take over the computation of other execution units, scheduling timing anomalies cannot occur in the MPC5xx processor pipeline. Domino effects like the one of MPC755 instruction scheduling mechanism [38] are thus not possible. Even though the processor employs programmable static branch prediction, speculation timing anomalies cannot occur due to the lack of an instruction cache.

However, all MPC56x processor derivatives (i.e., all MPC5xx processors except the MPC555) feature a branch target instruction cache (BTIC). The BTIC replaces entries in a first-in first-out (FIFO) manner, which is known to cause domino effects [3] in general. As we will demonstrate in the following, it is possible to construct a program that suffers from a domino effect caused by the FIFO replacement policy. For the domino effect to occur a non-LWC decision, such as a BTIC hit, is not a necessary precondition.

The PowerPC ISA comprises the following branch instructions: Branch `b`, conditional branch `bc`, and register indirect branches `blr`, `bctr`, with conditional versions `bclr`, `bcctr`. Unconditional branches and backward conditional branches are predicted taken, all others are predicted not-taken. For conditional branches this behavior can be reversed by changing a dedicated bit in the opcode. Branch prediction only takes place if the branch condition is still unevaluated and the target address is already known. For register indirect branches the target address is known if no other instruction to be executed before the branch writes to the respective register.

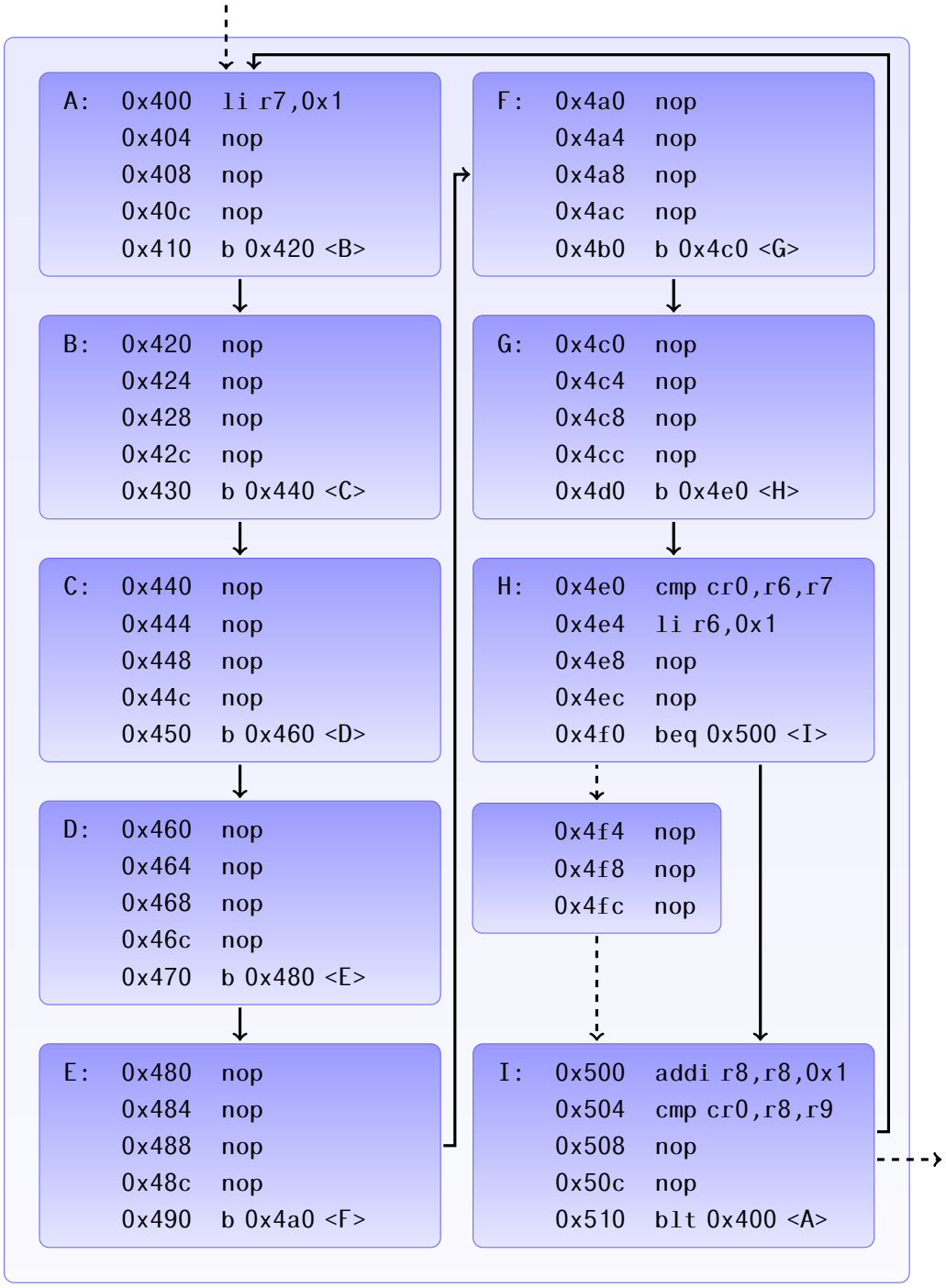



Figure 6.10: *Code Snippet exhibiting MPC56x Domino Effect:* Example program that suffers from a domino effect. Depending on the initial value of register r6, the branch target buffer will provide nine hits per two loop iterations (e.g., r6 = 0), or only misses (e.g., r6 = 1).

The BTIC can store up to four subsequent instructions for up to eight branch targets. It is fully-associative, which means that information for a branch may be stored in any of the eight lines. If the BTIC is full, lines are replaced in accordance with the FIFO replacement strategy. On a change of flow (COF), the BTIC is queried. If the BTIC contains information for this branch (i.e., the fetch request is a BTIC hit), the subsequent cached instructions are fetched out of the BTIC with a latency of one cycle each. If the BTIC does not contain information for this branch (a BTIC miss), the line whose contents are to be replaced next is freshly allocated. This line is then filled with the instructions that will be fetched next, e.g., from external memory with a latency between 2 and 60 cycles each (depending on the memory type). This filling process will be detailed later. The number of cached instructions in a valid line may vary from two to four. Caching starts with the instruction at the branch target and ends after four instructions or when another COF occurs. If only one instruction would be cached, the whole line is marked as invalid. In this case the FIFO counter is incremented nonetheless, i.e., the line is not immediately reused upon the next BTIC miss.

Figure 6.10 depicts a small program (i.e., a loop construct) that exhibits a domino effect caused by the FIFO replacement strategy. Dashed edges denote non-COF control-flow transitions. Non-dashed edges between the basic blocks represent COF transitions that cause an update of the BTIC. The loop repeats as long as the content of register r8 is less than the value of register r9. Initially we assume r8 to be zero and r9 to be an arbitrary natural number $n \in \mathbb{N}$ with $n > 1$. The first loop round is necessary to setup the initial BTIC state that depends on the initial contents of register r6. Starting from the second loop round every branch in the loop will be taken until the last loop iteration.

We start with an empty BTIC and assume the value of r6 to be initially one. Like in the second loop iteration, every branch inside the loop will be taken. In this case the BTIC is unable to produce any hit, because there are nine change-of-flow transitions per loop round but only eight branch target instruction cache entries. After each loop iteration the BTIC will contain the last eight branch targets $[A, I, H, G, F, E, D, C]$. The rightmost element denotes the entry that is to be replaced next.

Now assume r6 to be zero prior to the execution of the loop. Then the comparison of the registers r6 and r7 causes the branch at instruction 0x4f0 not to be taken. As there is no non-successor change-of-flow to 0x500 the BTIC will not cache the first four instructions of the basic block I. After the first loop iteration the BTIC then contains the branch targets $[A, H, G, F, E, D, C, B]$. In the second loop iteration the control-flow transitions to all basic blocks except I will cause BTIC hits. As Table 6.15 shows the BTIC provides nine hits per two loop iterations starting from the second loop round. This effect never stabilizes which proves the presence of a domino effect caused by the replacement policy of the BTIC.

| | r6 = 1 | | r6 = 0 | |
|---|---|---|---|---|
| Round | BTIC Contents | Misses | BTIC Contents | Misses |
| 0 | [] | 0 | [] | 0 |
| 1 | $[A,I,H,G,F,E,D,C]$ | 9 | $[A,H,G,F,E,D,C,B]$ | 8 |
| $2_A$ | $[B,A,I,H,G,F,E,D]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_B$ | $[C,B,A,I,H,G,F,E]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_C$ | $[D,C,B,A,I,H,G,F]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_D$ | $[E,D,C,B,A,I,H,G]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_E$ | $[F,E,D,C,B,A,I,H]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_F$ | $[G,F,E,D,C,B,A,I]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_G$ | $[H,G,F,E,D,C,B,A]$ | 1 | $[A,H,G,F,E,D,C,B]$ | 0 |
| $2_H$ | $[I,H,G,F,E,D,C,B]$ | 1 | $[I,A,H,G,F,E,D,C]$ | 1 |
| $2_I$ | $[A,I,H,G,F,E,D,C]$ | 1 | $[I,A,H,G,F,E,D,C]$ | 0 |
| 3 | $[A,I,H,G,F,E,D,C]$ | 9 | $[A,H,G,F,E,D,C,B]$ | 8 |
| 4 | $[A,I,H,G,F,E,D,C]$ | 9 | $[I,A,H,G,F,E,D,C]$ | 1 |
| 5 | $[A,I,H,G,F,E,D,C]$ | 9 | $[A,H,G,F,E,D,C,B]$ | 8 |

Table 6.15: *MPC56x BTIC Domino Effect:* Before the execution of the loop (see Figure 6.10) we start with an initially empty branch target instruction cache. The initial contents of register r6 effectuate different BTIC contents after the first loop round. During the second loop round the execution of the basic blocks (i.e., $2_A$, $2_B$, etc.) either provoke BTIC misses only or just a single BTIC miss. The impact on timing depending on the initial state of register r6 never stabilizes, proving the presence of a domino effect in the MPC56x hardware architecture.

In the above example the domino effect is effectively triggered by the unknown contents of register r6. Similarly, we could have assumed r6 to accept the value 1 and different initial contents of the branch target instruction cache to observe the same anomalous (timing) behavior. However, it is not safe to assume an empty BTIC or a BTIC that is filled with irrelevant data to achieve the worst-case behavior. Reineke [32] provides an example, where a non-empty initial cache state leads to a worse cache hit-miss ratio compared to an empty initial cache state if using the FIFO replacement strategy.

Due to these finding the MPC5xx hardware architecture has to be classified as non-compositional hardware architecture in general. However, the MPC555 processor and the MPC56x processor derivatives with disabled BTIC can be categorized as fully timing compositional. Neither the analysis of benchmark programs nor of industry-level software provided evidence for the presence of timing anomalies.

### 6.2.5 MPC55xx

To investigate the presence of timing anomalies in the MPC55xx hardware architecture (using the e200z6 processor core), we automatically processed over 2000 different test programs. The test suite comprises a variety of tests, such as simple benchmark programs, handcrafted measurement tests, cryptographic algorithms, automotive as well as avionics applications. Our observations are sobering. Depending on the configuration of the hardware, the MPC55xx processor offers many occasions for timing anomalies and even domino effects.

All of the observed anomalies are speculation timing anomalies caused by the prefetch mechanism or cache timing anomalies due to the cache replacement policies. We figure that scheduling anomalies cannot occur, due to the in-order execution pipeline of the MPC55xx architecture. Every instruction can only be allocated to a dedicated execution unit. If that execution unit is currently in use, the dispatch of the corresponding instruction is halted until the execution unit is available.

In the following we discuss three instances of speculation timing anomalies that have been detected automatically. Not all the timing-anomalous hardware behavior we observed by abstract program simulation can also happen in the real hardware. This is caused by the abstraction from the concrete hardware state and the abstract state join operator which induces an inevitable loss of information. Concluding this section we provide an example where we can observe some hardware behavior in the abstract domain that would never occur in reality.

#### ALU-triggered Timing Anomaly

Figure 6.11 depicts a small example program that exhibits a speculation anomaly that is caused by the processor's prefetch mechanism. The basic idea is that an initial code fetch causes a successive cache miss that evicts cached data that is used later on. In our example, an initial cache hit for the code fetch request to the first instruction of `func1` at `0x3f60` exactly triggers this behavior. While receiving two instructions per clock cycle from the cache, the processor triggers a prefetch of the adjacent cache line at `0x3f80`[12] before the branch unit is able to decide about the control-flow change at the call instruction `bl 0x3f40 <func2>`. In case the prefetch of `0x3f80` misses the cache, the cache might evict data at address `0x5f80` from the cache that is accessed in `func2`. Evicting that data causes an additional cache line fill operation. This additional line fill would not occur if the initial code request would have missed the cache. In that case, the branch unit would be able to change control flow earlier and thus prevent the unnecessary cache line fill.

[12] Cache lines of the MPC55xx hardware architecture are 32 bytes wide.

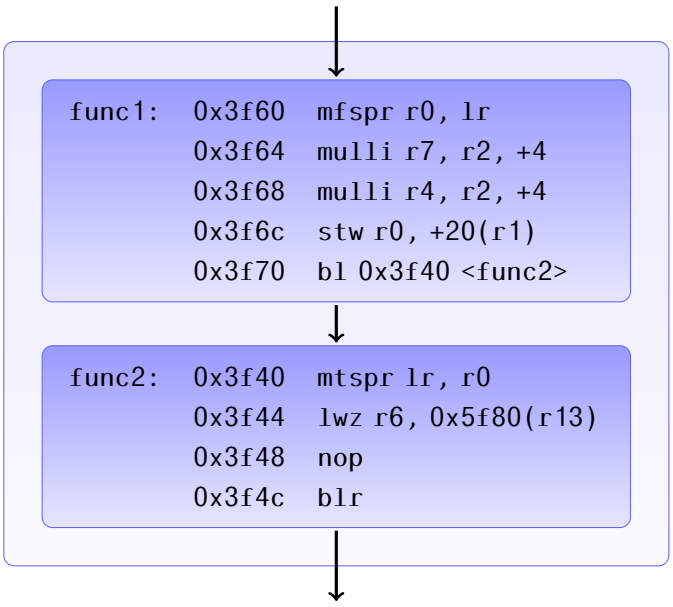



Figure 6.11: *MPC55xx ALU-triggered Timing Anomaly:* An initial cache hit to 0x3f60 triggers a prefetch of the unused successor cache line at 0x3f80. The branch unit is unable to redirect control flow because the instruction pipeline is busy executing the `mulli` instructions. The cache miss can evict a previously cached variable that is accessed in the function `func2`.

To experimentally verify this timing anomaly, we executed the example program on an MPC5554 evaluation board. The program code is executed from an external SRAM. The memory access timing is configured to zero cycle wait states. Cache line fill operations are split up into four-beat burst requests. Prior to the execution of the routine `func1` we ensure that the data at address 0x5f80 is already present in the cache. For the first measurement we pre-load the cache line starting at address 0x3f60 into the cache to provoke an initial cache hit. For the second experiment we ensure that this cache line is not cached initially.

Table 6.16 shows the measured bus events. *TS* denotes a transfer-start event, *TA* denotes a transfer-acknowledge event. The trace lines marked red indicate events that contribute to the global worst-case timing behavior following a non-LWC decision. Trace lines marked blue denote events that arise from an LWC behavior.

If the cache line at 0x3f60 is in the cache, the processor commences to prefetch the instructions at 0x3f80 before changing control-flow to `func2`. This cache miss then causes the eviction of the later accessed data at 0x5f80 from the unified cache. Missing the cache initially does not evict that data from the cache. Even though the processor takes longer to call `func2` (after 6 cycles instead of 3 cycles – the cache

| Initial Cache Hit | | | Initial Cache Miss | | |
|---|---|---|---|---|---|
| Event | Address | Cycles | Event | Address | Cycles |
| TS | 0x3f80 | 3 | **TS** | **0x3f60** | **3** |
| TA | 0x3f80 | 1 | **TA** | **0x3f60** | **1** |
| TA | 0x3f84 | 1 | **TA** | **0x3f64** | **1** |
| TA | 0x3f88 | 1 | **TA** | **0x3f68** | **1** |
| TA | 0x3f8c | 1 | **TA** | **0x3f6c** | **1** |
| TS | 0x3f90 | 1 | **TS** | **0x3f70** | **1** |
| TA | 0x3f90 | 1 | **TA** | **0x3f70** | **1** |
| TA | 0x3f94 | 1 | **TA** | **0x3f74** | **1** |
| TA | 0x3f98 | 1 | **TA** | **0x3f78** | **1** |
| TA | 0x3f9c | 1 | **TA** | **0x3f7c** | **1** |
| TS | 0x3f40 | 3 | TS | 0x3f40 | 6 |
| TA | 0x3f40 | 1 | TA | 0x3f40 | 1 |
| TA | 0x3f44 | 1 | TA | 0x3f44 | 1 |
| TA | 0x3f48 | 1 | TA | 0x3f48 | 1 |
| TA | 0x3f4c | 1 | TA | 0x3f4c | 1 |
| TS | 0x3f50 | 1 | TS | 0x3f50 | 1 |
| TA | 0x3f50 | 1 | TA | 0x3f50 | 1 |
| TA | 0x3f54 | 1 | TA | 0x3f54 | 1 |
| TA | 0x3f58 | 1 | TA | 0x3f58 | 1 |
| TA | 0x3f5c | 1 | TA | 0x3f5c | 1 |
| **TS** | **0x5f80** | **2** | | | |
| **TA** | **0x5f80** | **1** | | | |
| **TA** | **0x5f84** | **1** | | | |
| **TA** | **0x5f88** | **1** | | | |
| **TA** | **0x5f8c** | **1** | | | |
| **TS** | **0x5f90** | **1** | | | |
| **TA** | **0x5f90** | **1** | | | |
| **TA** | **0x5f94** | **1** | | | |
| **TA** | **0x5f98** | **1** | | | |
| **TA** | **0x5f9c** | **1** | | | |
| Total | | 35 | | | 27 |

Table 6.16: *MPC55xx Measurement for ALU-triggered Anomaly:* If the first code fetch of the example program Figure 6.11 hits the cache, an additional cache line is prefetched while executing the `mulli` instructions. This cache miss evicts a previously cached variable that is accessed in `func2`.

line fill operation partially subsumes the execution time of the call instruction), it thus does not require to reload the entire cache line. Hence, the local worst-case assumption, i.e., the first cache line is not resident in the unified cache, does not lead to the global worst-case execution behavior. Assuming an initial cache hit, the processor takes 35 cycles to execute the program. Otherwise, the program completes after 27 cycles.

### LSU-triggered Timing Anomaly

Figure 6.12 depicts a small program where we could observe a similar execution behavior. Depending on whether the memory access of the `lwz` instruction at address `0x4f4` hits the cache, the processor performs a superfluous code prefetch that slows down the overall execution of the program. A special hardware configuration is required to trigger this speculation anomaly.

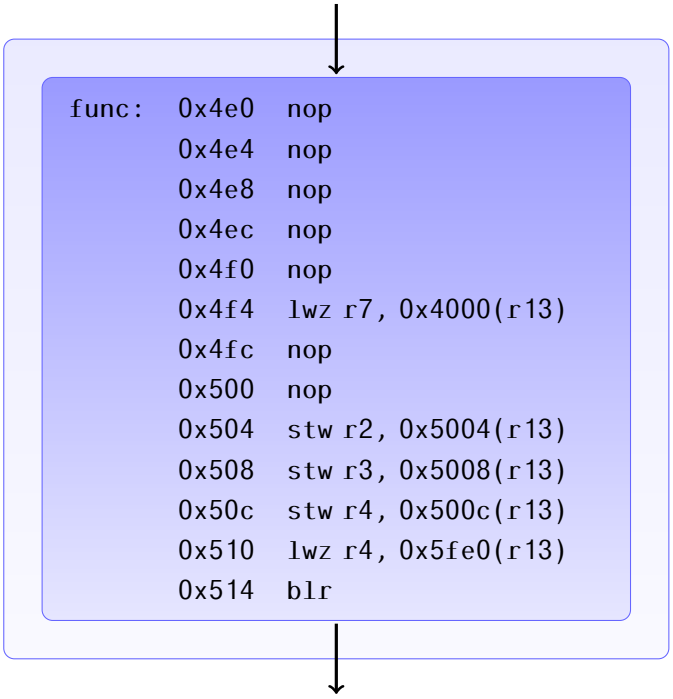



Figure 6.12: *MPC55xx LSU-triggered Timing Anomaly:* This example program executes slower if the first data access (i.e., `lwz` instruction at address `0x4f4`) hits the unified cache. In this case, the processor will initiate a code prefetch that causes an additional cache line fill delaying the last two data accesses of the program.

To observe the discussed speculation timing anomaly we configure an MPC5554 evaluation board such that only code fetches can trigger cache line fill operations.

| Cache Hit | | | Cache Miss | | |
|---|---|---|---|---|---|
| Event | Address | Cycles | Event | Address | Cycles |
| TS | 0x0500 | 3 | TS | 0x0500 | 3 |
| TA | 0x0500 | 1 | TA | 0x0500 | 1 |
| TA | 0x0504 | 1 | TA | 0x0504 | 1 |
| TA | 0x0508 | 1 | TA | 0x0508 | 1 |
| TA | 0x050c | 1 | TA | 0x050c | 1 |
| TS | 0x0510 | 1 | TS | 0x0510 | 1 |
| TA | 0x0510 | 1 | TA | 0x0510 | 1 |
| TA | 0x0514 | 1 | TA | 0x0514 | 1 |
| TA | 0x0518 | 1 | TA | 0x0518 | 1 |
| TA | 0x051c | 1 | TA | 0x051c | 1 |
| | | | **TS** | **0x4000** | **2** |
| | | | **TA** | **0x4000** | **1** |
| TS | 0x5004 | 4 | TS | 0x5004 | 4 |
| TA | 0x5004 | 1 | TA | 0x5004 | 1 |
| TS | 0x5008 | 2 | TS | 0x5008 | 2 |
| TA | 0x5008 | 1 | TA | 0x5008 | 1 |
| **TS** | **0x0520** | **2** | | | |
| **TA** | **0x0520** | **1** | | | |
| **TA** | **0x0524** | **1** | | | |
| **TA** | **0x0528** | **1** | | | |
| **TA** | **0x052c** | **1** | | | |
| **TS** | **0x0530** | **1** | | | |
| **TA** | **0x0530** | **1** | | | |
| **TA** | **0x0534** | **1** | | | |
| **TA** | **0x0538** | **1** | | | |
| **TA** | **0x053c** | **1** | | | |
| TS | 0x500c | 2 | TS | 0x500c | 2 |
| TA | 0x500c | 1 | TA | 0x500c | 1 |
| TS | 0x5fe0 | 2 | TS | 0x5fe0 | 2 |
| TA | 0x5fe0 | 1 | TA | 0x5fe0 | 1 |
| Total | | 37 | | | 29 |

Table 6.17: *MPC55xx Measurement for LSU-triggered Anomaly:* Measured events that prove the presence of a speculation anomaly in the MPC5554 processor. If the first load access hits the cache, the processor prefetches an additional cache line that delays pending memory accesses.

For this purpose we modify the L1 cache control and status register 0 (L1CSR0) to disable every way of the unified cache for data miss fills.

As before, the program code is executed from an external SRAM module. The external memory access timing is configured to zero cycle wait states. Cache line fill operations targeting the external memory are split up into four-beat burst requests. The remainder of the hardware configuration is left unmodified.

Again we perform two experiments. For the first one, we ensure that the first data access to 0x4000 is going to hit the cache. For the second one, we enforce every data access to miss the cache. Due to the cache configuration, the processor does then only perform single-beat accesses to load the requested data from external memory.

Table 6.17 displays the measured bus events for both experiments. Due to space limitations the table only shows the observed events starting with the cache line fill operation that loads the remainder of routine `func`. If the first data access hits the cache, the prefetch queue is emptied faster, because the load access does not block the execution. To prevent the processor from waiting for incoming instructions, the fetch unit initiates subsequent code prefetches that cause an additional cache line fill (i.e., starting at address 0x520). This delays the last two memory accesses by the number of processor cycles it takes to perform the cache line operation. Executing the remainder of the routine `func` completes after 37 cycles.

Otherwise, if the first data access misses the cache, the processor stops the prefetch in favor of the pending store accesses. The remainder of the routine `func` then finishes after 29 cycles. Hence assuming the `lwz` instruction to miss the unified cache does not affect the worst-case execution behavior.

### FLASH Prefetch Timing Anomaly

We also found a scheduling timing anomaly that is related to the prefetch mechanism of the internal FLASH memory module. Similar to the unified cache, the FLASH memory is organized into pages of 32 bytes each.[13] To speed up execution, the FLASH memory controller comprises a two-page prefetch buffer. The controller can be furthermore configured to prefetch the neighboring page if the last doubleword is being accessed.

Figure 6.13 depicts a minimal program that triggers a timing anomaly depending on the state of the FLASH controller. For this example we disable the cache and configure the FLASH to prefetch the neighboring page if the last doubleword of the previous one is accessed. The number of FLASH read wait states is configured to seven processor cycles. Furthermore we disable the store buffer, to prevent

[13] A cache line is of the same size.

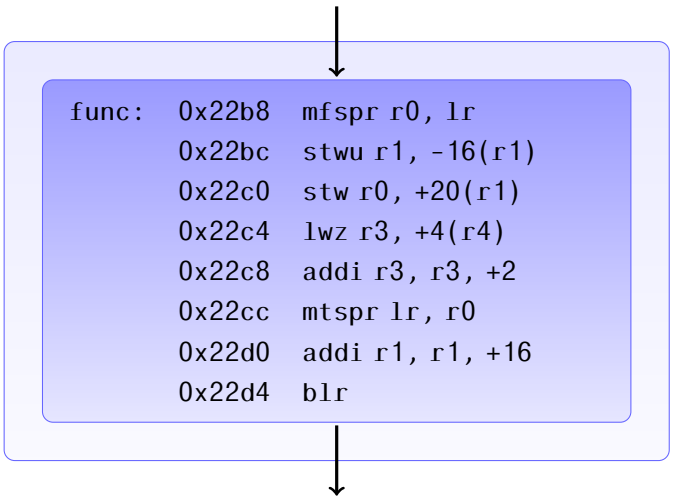



Figure 6.13: *MPC55xx FLASH Prefetch Timing Anomaly:* The internal FLASH memory prefetch mechanism causes a timing anomaly. If the FLASH page starting at 0x22c0 has been prefetched earlier, the SRAM store accesses will be executed in a back to back manner, which induces a two-cycle access penalty.

stores from being deferred. The remainder of the hardware configuration is left unmodified.

Depending on the code executed prior to the function `func` the FLASH prefetch buffers might not contain the page that is currently accessed (i.e., starting from 0x22a0), but its neighboring page (i.e., starting from 0x22c0). After fetching the first two instructions from the internal FLASH memory, the decode queue is quickly filled up because the successor FLASH page is already prefetched. The processor then stops the code prefetch in favor of the pending data accesses, which are then executed in a back to back manner. Due to this processor state, the two store instructions (i.e., `stwu` and `stw`) access the internal SRAM one ofter another. Executing a 32bit store access right after another 32bit store access has completed incurs a two-cycle penalty (see MPC5554 reference manual Table 15-2 [28]). In total, the program snippet terminates after 62 cycles.

The SRAM penalty does not apply in case the FLASH prefetch buffers neither contains the first FLASH page nor its successor. Even though the instruction fetches then take longer to complete, the overall execution is not additionally delayed. The two store accesses are not issued consecutively, because the first store will already have left the instruction pipeline after the second one is put into the decode queue. Hence, the SRAM access penalty does not apply and the routine executes after 60 cycles.

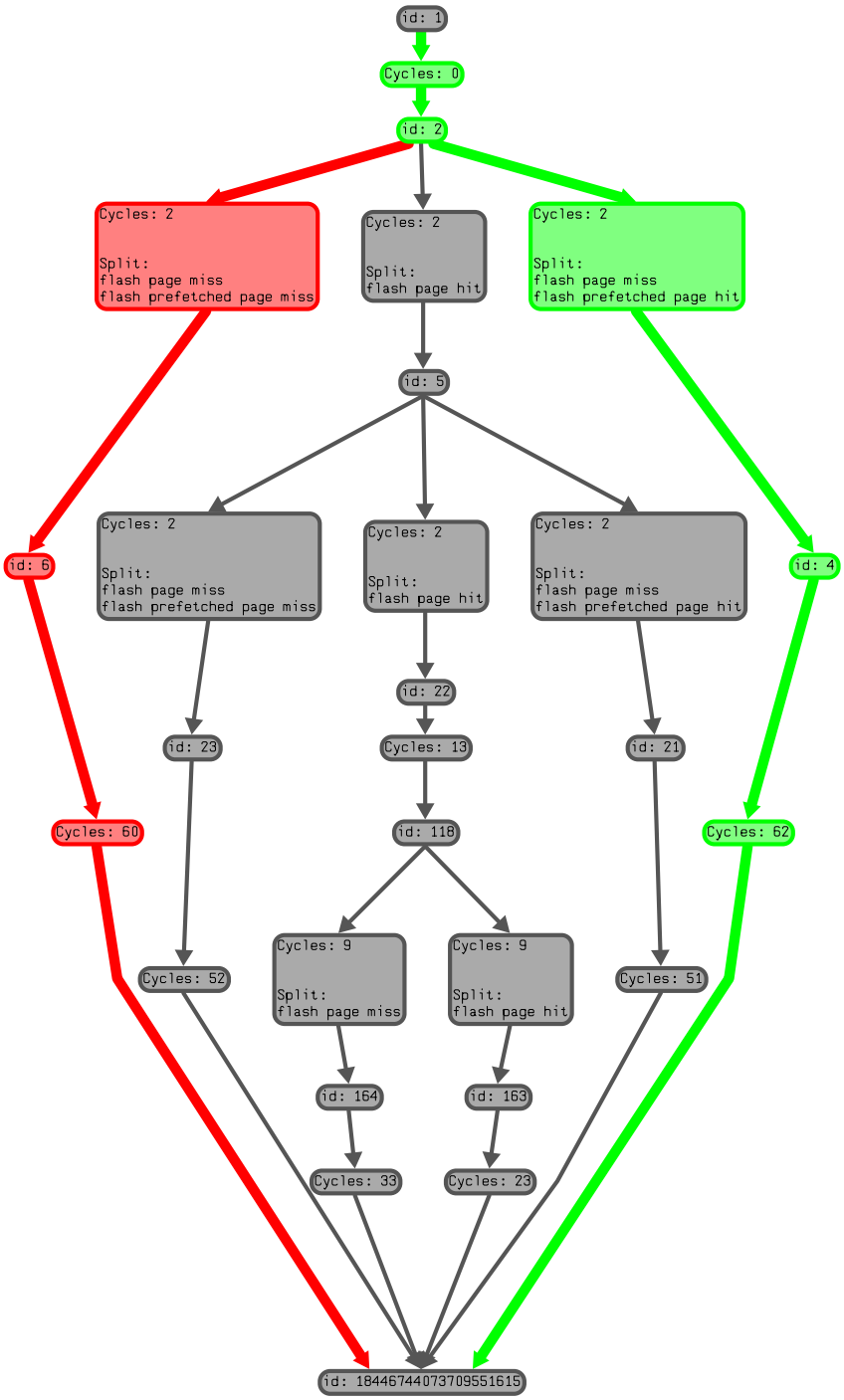



Figure 6.14: *Visualization of MPC55xx FLASH Prefetch Timing Anomaly:* Considering only local worst-case decisions (red path) does not contribute to the global worst-case execution behavior (green path).

Figure 6.14 depicts the obtained prediction graph from the abstract program simulation. The green path denotes the worst-case execution path on which a non-LWC decision takes place. Only considering the local worst-case avoids the second store to the internal SRAM to wait, which hence does not contribute to the global worst-case behavior.

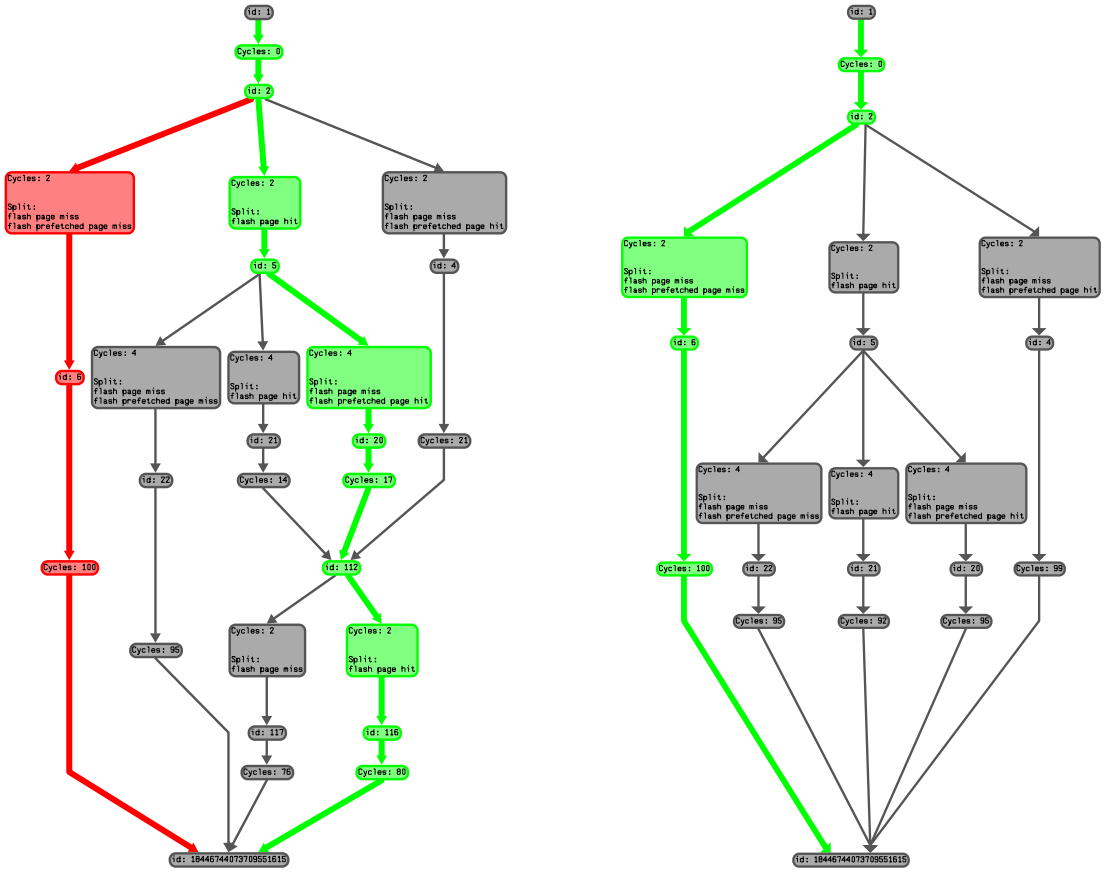



Figure 6.15: *MPC55xx Virtual Timing Anomaly:* The left prediction graph shows a possible timing anomaly, where the initial FLASH read buffer hit leads to the worst-case execution behavior. Forbidding state joins, the pipeline analysis produces the prediction graph on the right-hand side. In that graph the supposed timing anomaly is revealed as false positive.

### Virtual Timing Anomaly

Not every instance of a timing anomaly we could automatically detect is necessarily a genuine one. By nature, abstract interpretation introduces a loss of information as discussed throughout Chapter 2. This causes the pipeline analysis to assume a hardware behavior that would not occur during any execution on the real device.

For example the abstract state join operator or the widening operator can cause such an information loss.

Figure 6.15 exemplifies this issue by means of two prediction graphs. As indicated by the red path, the left prediction graph exhibits a timing anomaly that is caused by an initial FLASH read buffer hit. The program is estimated to execute in 103 cycles. However, if we disallow any cache and buffer state joins during the pipeline analysis as shown in the right prediction graph, the timing anomaly instance cannot be established. The estimated worst-case execution is determined to take 102 processor cycles. In effect, the state join (see left prediction graph) causes the pipeline analysis to assume an execution behavior that actually would not occur in reality. In this particular case, the analysis has lost information about the precise contents of the internal FLASH read buffers. As a consequence, the pipeline analysis will then later split, because it is unable to decide whether a later access to the FLASH memory hits or misses the FLASH read buffers.

Clearly, the occurrence of virtual timing anomalies cannot be entirely avoided. State joins are necessary to enable a static WCET analysis of complex programs (and hardware architectures). Even though this causes a loss of worst-case performance, we found that the impact of virtual timing anomalies on the WCET estimation is rather minimal for the MPC55xx hardware architecture.

#### Domino Effects

Similarly to the MPC56x processor derivatives, the MPC55xx (e200z6 core) hardware architecture is equipped with an eight-entry branch target buffer (BTB). But contrary to the MPC56x implementation, the BTB only stores the branch target and additionally a two-bit history counter. The counter is incremented if the branch was taken and decremented otherwise. In this fashion the BTB entry indicates how likely the branch is to be taken. Depending on the value of the history counter the MPC55xx core speculatively prefetches in either taken or non-taken direction. Entries of the branch target buffer are updated according to the FIFO principle. It is possible to construct a program similar to Figure 6.10 that exploits a domino effect due to the FIFO replacement policy.

The replacement policy of the unified cache also opens the possibility for domino effects. Cache lines are replaced by means of a pseudo-round-robin replacement algorithm. A global replacement counter indicates the cache line to be replaced next. In principle the round-robin replacement strategy behaves like the FIFO replacement policy. If we only consider a single cache set, the global replacement counter causes a FIFO update behavior. Hence, we can easily construct a program that exploits a domino effect due to the behavior of the replacement policy.

#### Summary

The above findings indicate that the MPC55xx (e200z6) is a hardware architecture that offers many possibilities for timing anomalies to occur. Unfortunately, it is hardly possible to locally decide about whether the pipeline analysis has to consider a non-LWC decision in order to compute a safe WCET bound. The future behavior of the executed program decides about the global impact of a timing anomaly instance. It might as well be that the slower execution of a routine caused by prefetching a piece of code that is not needed immediately turns out to be beneficial in the long run due to pre-caching instructions that are later begin executed.

We furthermore state that the MPC55xx hardware offers (at least) two disjoint hardware features that exhibit domino effects. In either case the FIFO replacement strategy causes this kind of anomalous execution behavior.

On the bottom line, we classify the MPC55xx architecture as non-compositional hardware architecture. Nonetheless the hardware can be configured to avoid the occurrence of domino effects after all by disabling the BTB and locking all except one cache way. Disabling the BTB and locking all cache ways except

### 6.2.6 Intel 386

We automatically processed about 1500 test programs to investigate the existence of timing anomalies in the Intel 386 hardware architecture. The available tests comprise rather simple benchmark programs and several tasks from an avionics application. In fact we could find some instances of timing anomalies. However, a closer look reveals that all the timing anomalies we have detected are in fact virtual. At some point the pipeline analysis is unsure about the processor state, such as the stalling of code fetches for example, and follows both possibilities.

Figure 6.16 depicts one instance of these virtual timing anomalies. In the depicted prediction graph the pipeline analysis is unsure whether the code fetch is stalled due to the fill level of the decode queue. This behavior strongly depends on the micro-code for the corresponding fetched instruction. Depending on the size of the micro-code the processor stalls the prefetch to prevent a micro-code buffer overflow. Precise information about the instruction micro-code is unfortunately not available, which is why the pipeline analysis is designed to investigate both a stalled and an ongoing prefetch. Note that this analysis behavior is only a documentation problem. By means of precise information about the micro-code handling, this analysis would not assume hardware behavior that does not occur in reality. The prediction graph reveals that assuming the code fetch to stall in this special situation (see red path) would not contribute to the global worst-case behavior (see green path). Assuming the prefetch to continue in the first place causes the pipeline to again decide whether

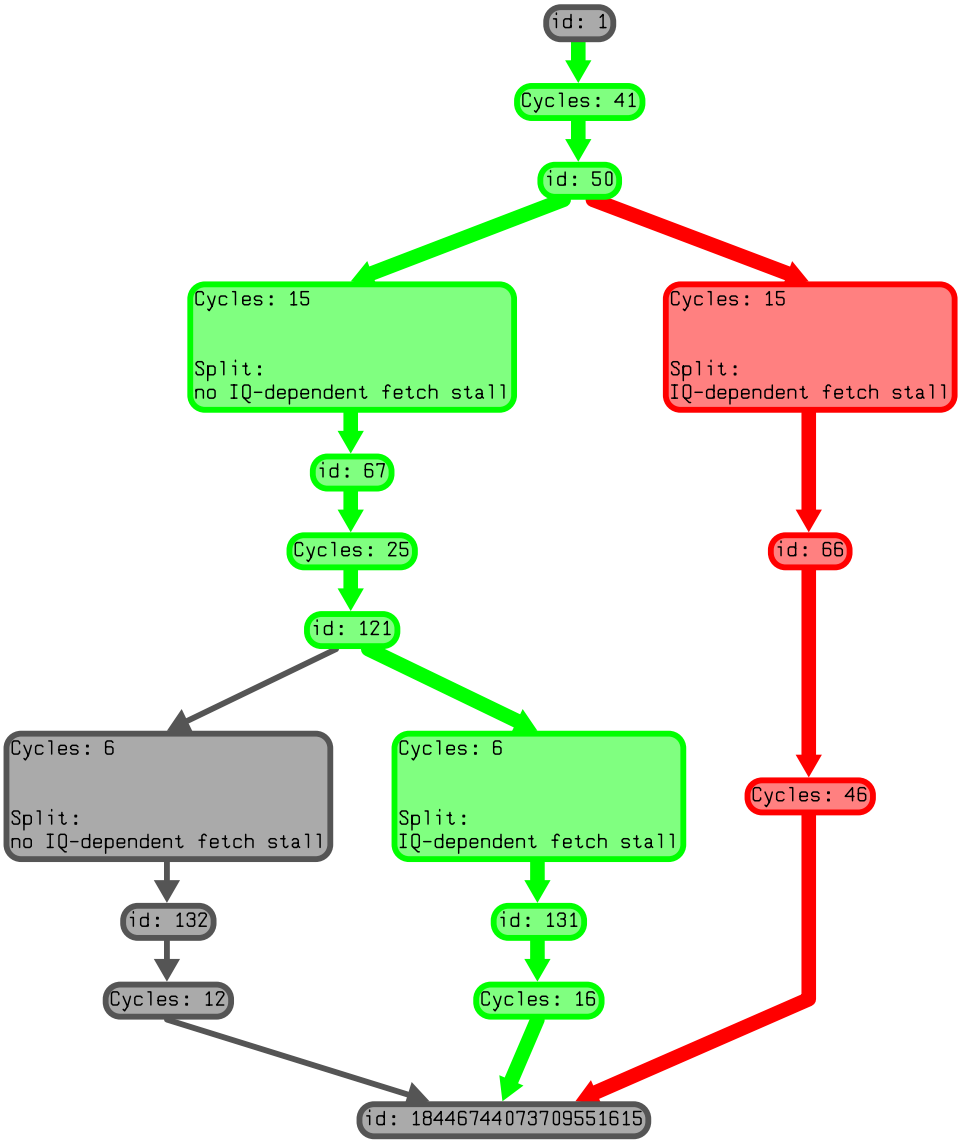



Figure 6.16: *I386 Virtual Timing Anomaly:* The pipeline analysis is unsure about whether the code fetch is stalled. Depending on the fill level of the decode queue, the processor stalls to prevent an internal micro-code buffer overflow. Information about the decoding behavior of the micro-code is not publicly available. Thus, the analysis has to assume both possibilities, which provokes a virtual timing anomaly in this case.

the code fetch is stalled. The local worst-case decision will then effectuate the globally worst execution behavior.

This kind of timing anomaly cannot occur in reality because the instruction decoding is deterministic and the micro-code for the instructions are fixed. Starting from the same initial hardware state, it is not possible to effectuate a filling of the decode queue for the sequence of instructions where it is unclear whether the prefetch stalls for whatever reason. The sole unknown that causes this decision during the static pipeline analysis is the behavior of the processor-internal micro-code, for which there is no official documentation available.

In total we observed instances of these virtual timing anomalies for about 23% of the available test programs. Their impact on the precision of the analysis is however strictly limited as the trace validation results show (see Table 6.11 on page 102). We could not find any evidence for the occurrence of real timing anomalies. Because of the rather simple structure of the Intel 386 hardware architecture (see Figure 6.6 on page 100) this is not unexpected. The processor does not perform any kind of branch prediction. Without any exception, the Intel 386 predicts all branches as not taken. This avoids the occurrence of speculation anomalies. We can also safely exclude the presence of scheduling anomalies because instructions are always executed in order. The processor does not feature any cache and is thus unable to provoke any kind of cache timing anomaly. Based on this investigation we classify the Intel 386 as a fully timing compositional hardware architecture.

### 6.2.7 AMD 486

For the exploration of AMD 486 timing anomalies, we only had about 300 different test programs at hand. The majority of the available tests comprise avionics applications. Prior to the investigation we did not expect scheduling timing anomalies to occur. As Figure 6.7 shows, the AMD 486 features two disjoint execution units – the ALU for integer arithmetic and the FPU for floating-point computation. Even though a (limited) form of execution parallelism exists, the executions units cannot impact the timing behavior of each other such that a timing anomaly would occur. A faster execution in one unit is unable to cause a slower execution in the other unit. Our observations confirm this assumption.

During the investigation, we could identify an instance of a speculation anomaly that is caused by a data access that may hit the on-chip unified cache. Figure 6.17 depicts a code snippet by means of which we could observe a timing anomaly. Before the CPU is able to change the control-flow to the branch target 0x1730, the processor first has to perform the memory accessing instruction at 0x1709 and the successive instructions. If the data access hits the cache, the processor decides to prefetch the cache line starting at address 0x1720. Unable to abort the prefetch, the execution of

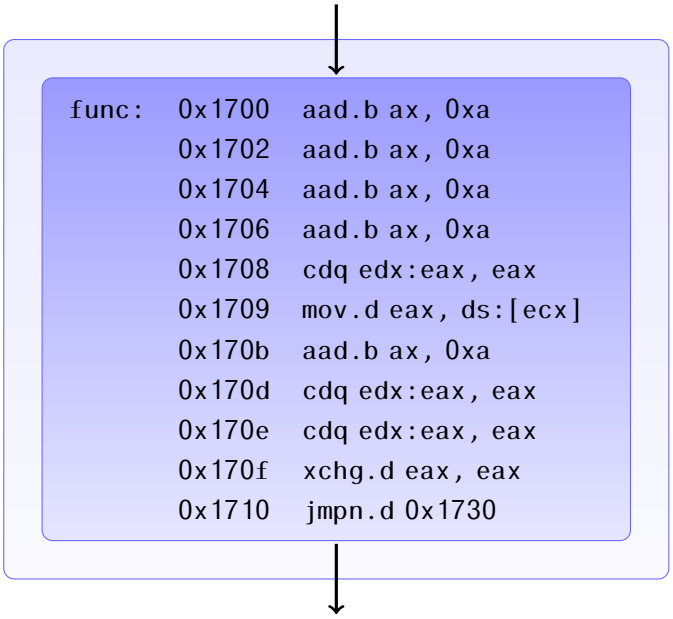



Figure 6.17: *AMD 486 Speculation Timing Anomaly:* The unified cache can cause a timing anomaly. If the memory access at 0x1709 hits the cache, the processor will perform an additional prefetch of the cache line starting at 0x1720. This prefetch lengthens the overall execution and the processor has to wait longer before redirecting the control-flow to the branch target.

the branch instruction at 0x1710 is additionally delayed. In total the code snippet executes in 162 cycles. But if the data access would miss the cache, the processor is able to execute the remaining instructions and decide early about the control-flow while performing the cache line fill operation. In this case, the program executes in 144 cycles only. Table 6.18 depicts the measured bus events and hence proves the existence of this speculation timing anomaly.

In addition to the above finding, the AMD 486 processor suffers from a cache timing anomaly. The unified on-chip cache uses a PLRU cache replacement policy that is known to trigger domino effects [3]. Similar to the PLRU implementation of the Freescale MPC755 cache, the AMD 486 uses a Tree-PLRU algorithm that uses a binary search tree to determine the element to be replaced next. Hence, it is possible to construct a program that exhibits a domino effect.

Due to these observations the AMD 486 is classified as non-compositional hardware architecture. However, as the cache allows for cache locking, e.g., of frequently used code and data, it is possible to avoid the PLRU domino effect. This allows for a classification as compositional architecture.

| Cache Hit | | | Cache Miss | | |
|---|---|---|---|---|---|
| Event | Address | Cycles | Event | Address | Cycles |
| TS | 0x1700 | 9 | TS | 0x1700 | 9 |
| TS | 0x1704 | 9 | TS | 0x1704 | 9 |
| TS | 0x1708 | 9 | TS | 0x1708 | 9 |
| TS | 0x170c | 9 | TS | 0x170c | 9 |
| TS | 0x1710 | 9 | TS | 0x1710 | 9 |
| TS | 0x1714 | 9 | TS | 0x1714 | 9 |
| TS | 0x1718 | 9 | TS | 0x1718 | 9 |
| TS | 0x171c | 9 | TS | 0x171c | 9 |
| | | | **TS** | **0x1760** | **9** |
| | | | **TS** | **0x1764** | **9** |
| **TS** | **0x1720** | **27** | **TS** | **0x1768** | **9** |
| **TS** | **0x1724** | **9** | **TS** | **0x176c** | **9** |
| **TS** | **0x1728** | **9** | TS | 0x1790 | 9 |
| **TS** | **0x172c** | **9** | TS | 0x1794 | 9 |
| TS | 0x1790 | 9 | TS | 0x1798 | 9 |
| TS | 0x1794 | 9 | TS | 0x179c | 9 |
| TS | 0x1798 | 9 | | | |
| TS | 0x179c | 9 | | | |
| Total | | 162 | | | 144 |

Table 6.18: *AMD 486 Measurement for Speculation Anomaly:* Measured transfer start events for the code snippet depicted in Figure 6.17. The processor prefetches far beyond the border of the executed basic block. If the data access hits the cache, the processor prefetches the unused cache line starting at 0x1720. This delays the execution of the branch instruction and the program executes in 162 processor cycles. Otherwise if the data access to 0x1760 misses the cache, the code snippet executes in only 144 processor cycles.

# 7 Conclusion

Determining safe and precise bounds on the worst-case execution time of safety-critical software for modern architectures is a major challenge. It is important to better understand the nature of timing anomalies and domino effects and their counter-intuitive behavior.

Our new semi-automatic method that is able to detect potential timing anomalies using an abstract interpretation based over-approximation of all possible hardware behaviors during execution of a program. From the results a test program can be built that show the timing effect of an anomaly on real processors.

With the new method we have found and measured yet unknown timing anomalies of the MPC55xx and AMD 486 hardware architectures. We have applied our new method to seven hardware architectures and provided a corresponding classification.

Timing models are typically handwritten in an error-prone and tedious process. We have also presented an automatic method to validate timing models of processors. Our new trace validation method automatically compares the behavior of an abstract hardware model against real execution traces and efficiently reveals issues with the model.

With a set of traces it is possible to established a necessary level of confidence in the correctness of the model as required by current safety standards like DO-178B. Our trace validation methodology has been successfully employed to validate various timing models of the industrial-strength worst-case execution time analyzer aiT.

# List of Tables

# List of Figures

# List of Theorems

# List of Algorithms

# Bibliography


[1] AEROFLEX GAISLER. http://www.gaisler.com.

[2] ALTMEYER, S., DAVIS, R. I., AND MAIZA, C. Cache related pre-emption aware response time analysis for fixed priority pre-emptive systems. In *Proceedings of the 32nd IEEE Real-Time Systems Symposium (RTSS'11)* (December 2011), R. I. Davis and N. Fisher, Eds., pp. 261–271.

[3] BERG, C. PLRU cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dresden* (July 2006), F. Mueller, Ed., no. 06902 in Dagstuhl Seminar Proceedings, Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI).

[4] CASSEZ, F., HANSEN, R. R., AND OLESEN, M. C. What is a Timing Anomaly? In *12th International Workshop on Worst-Case Execution Time Analysis* (Dagstuhl, Germany, 2012), T. Vardanega, Ed., vol. 23 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 1–12.

[5] COUSOT, P., AND COUSOT, R. Static determination of dynamic properties of programs. In *Proceedings of the second international Symposium on programming* (1976), Dunod, Paris, France, pp. 106–130.

[6] COUSOT, P., AND COUSOT, R. 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* (New York, NY, USA, 1977), ACM, pp. 238–252.

[7] DAVEY, B. A., AND PRIESTLEY, H. A. *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press, May 2002.

[8] EISINGER, J., POLIAN, I., BECKER, B., THESING, S., WILHELM, R., AND METZNER, A. Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In *DDECS '06: Proceedings of the 2006 IEEE Design and Diagnostics of Electronic Circuits and systems* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 13–18.

[9] ENGBLOM, J., AND JONSSON, B. Processor pipelines and their properties for static WCET analysis. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software* (London, UK, 2002), Springer-Verlag, pp. 334–348.

[10] ERC32 chipset documentation. http://microelectronics.esa.int/erc32/.

[11] FEDERAL AVIATION ADMINISTRATION. What is a "decision" in application of modified condition/decision coverage (MC/DC) and decision coverage (DC)? Position paper CAST-10, U.S. Department of Transportation, Washington, DC 20591, June 2002.

[12] FERDINAND, C. *Cache Behaviour Prediction for Real-Time Systems*. PhD thesis, Universität des Saarlandes, 1997.

[13] FERDINAND, C., AND WILHELM, R. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems 17*, 2-3 (1999), 131–181.

[14] GEBHARD, G., CULLMANN, C., AND HECKMANN, R. Software Structure and WCET Predictability. In *Bringing Theory to Practice: Predictability and Performance in Embedded Systems* (Dagstuhl, Germany, 2011), P. Lucas, L. Thiele, B. Triquet, T. Ungerer, and R. Wilhelm, Eds., vol. 18 of *OpenAccess Series in Informatics (OASIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 1–10.

[15] GRUND, D., REINEKE, J., AND GEBHARD, G. Branch target buffers: WCET analysis framework and timing predictability. In *15th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2009* (August 2009).

[16] INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS. VHDL language reference manual. IEEE Standard P1076 2000/D3, New York, 2000.

[17] The Kate Editor. http://www.kate-editor.org.

[18] KIRNER, R., KADLEC, A., AND PUSCHNER, P. Precise worst-case execution time analysis for processors with timing anomalies. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems* (Dublin, Ireland, July 2009), IEEE, pp. 119–128.

[19] LaTeX - A Document Preparation System. http://www.latex-project.org.

[20] LEON2 processor user's manual. http://www.arl.wustl.edu/.../leon2-1_0_23-xst.pdf.

[21] LIONS, J.-L., LÜBECK, L., FAUQUEMBERGUE, J.-L., KAHN, G., KUBBAT, W., LEVEDAG, S., MAZZINI, L., MERLE, D., AND O'HALLORAN, C. Ariane 5 flight 501 failure. Ariane 501 inquiry board report, ESA, Paris, Juli 1996.

[22] Longest path problem. http://en.wikipedia.org/wiki/Longest_path_problem.

[23] LUCAS, P., PARSHIN, O., AND WILHELM, R. Operating mode specific WCET analysis. In *Proceedings of JRWRTC* (October 2009), C. Seidner, Ed.

[24] LUNDQVIST, T., AND STENSTRÖM, P. Timing anomalies in dynamically scheduled microprocessors. In *Real-Time Systems Symposium (RTSS)* (December 1999).

[25] MAHAPATRA, N. R., AND VENKATRAO, B. The processor-memory bottleneck: Problems and solutions. *Crossroads - Computer architecture Crossroads Homepage archive 5*, 3es (Spring 1999).

[26] MARTIN, F., ALT, M., WILHELM, R., AND FERDINAND, C. Analysis of loops. In *Compiler Construction*, K. Koskimies, Ed., vol. 1383 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1998, pp. 80–94. 10.1007/BFb0026424.

[27] MOORE, G. Cramming more components onto integrated circuits. *Eletronics 19*, 3 (April 1965), 114–117.

[28] MPC5553 and MPC5554 microcontroller reference manual. http://freescale.com/files/.../MPC5553_MPC5554_RM.pdf.

[29] The Nexus 5001 Forum Standard for a Global Embedded Processor Debug Interface, 2003.

[30] NIELSON, F., NIELSON, H. R., AND HANKIN, C. *Principles of Program Analysis*, 2nd ed. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[31] PETTERS, S. M. Comparison of trace generation methods for measurement based WCET analysis. In *3rd International Workshop on Worst Case Execution Time Analysis* (Porto, Portugal, July 2003), pp. 75–78.

[32] REINEKE, J. *Caches in WCET Analysis*. PhD thesis, Universität des Saarlandes, November 2008.

[33] REINEKE, J., AND SEN, R. Sound and efficient wcet analysis in the presence of timing anomalies. In *9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis* (Dagstuhl, Germany, 2009), N. Holsti, Ed., Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[34] REINEKE, J., WACHTER, B., THESING, S., WILHELM, R., POLIAN, I., EISINGER, J., AND BECKER, B. A definition and classification of timing anomalies. In *Workshop on Worst-Case Execution-Time Analysis (WCET)* (July 2006).

[35] ROJAS, R. Konrad Zuse's legacy: The architecture of the Z1 and Z3. *IEEE Annals of the History of Computing 19*, 2 (April 1997), 5–16.

[36] Rubber - A Wrapper for LaTeX and Friends. http://launchpad.net/rubber.

[37] SCHLICKLING, M., AND PISTER, M. Semi-automatic derivation of timing models for WCET analysis. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems* (April 2010), ACM, pp. 67–76.

[38] SCHNEIDER, J. *Combined Schedulability and WCET Analysis for Real-Time Operating Systems*. PhD thesis, Saarland University, 2003.

[39] SHARIR, M., AND PNUELI, A. Two Approaches to Interprocedural Data Flow Analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, 1981, ch. 7.

[40] TC1797 microcontroller reference manual. http://www.infineon.com.

[41] THEILING, H. Ilp-based interprocedural path analysis. In *EMSOFT* (2002), A. L. Sangiovanni-Vincentelli and J. Sifakis, Eds., vol. 2491 of *Lecture Notes in Computer Science*, Springer, pp. 349–363.

[42] THESING, S. *Safe and Precise WCET Determinations by Abstract Interpretation of Pipeline Models*. PhD thesis, Saarland University, 2004.

[43] THOMAS, D., AND MOORBY, P. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Boston, Massechusetts, 1991.

[44] Topological sorting. http://en.wikipedia.org/wiki/Topological_sorting.

[45] WENZEL, I., KIRNER, R., PUSCHNER, P., AND RIEDER, B. Principles of timing anomalies in superscalar processors. In *Proceedings of the Fifth International Conference on Quality Software* (Washington, DC, USA, 2005), QSIC '05, IEEE Computer Society, pp. 295–306.

[46] WERNER, T., AND AKELLA, V. Asynchronous processor survey. *IEEE Computer 30*, 11 (1997), 67–76.

[47] WILHELM, R., GRUND, D., REINEKE, J., SCHLICKLING, M., PISTER, M., AND FERDINAND, C. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems 28*, 7 (July 2009), 966–978.

# Index

## Symbols



## A



## B



## C



## D



## E



## F



## I



## L



## M



## N

## O



## P



## S



## T



## U



## V