**Precision Timed Machines**

by

Isaac Liu

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:
Professor Edward A. Lee, Chair
Professor John Wawrzynek
Professor Alice Agogino

Spring 2012

The dissertation of Isaac Liu, titled Precision Timed Machines is approved:

Chair Date

Date

Date

University of California, Berkeley

Spring 2012

**Precision Timed Machines**

# Abstract

Precision Timed Machines

by

Isaac Liu
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

This is my abstract

To my wife, my parents, and everyone else whom I've had the privilege of running into for the first twenty-seven years of my life.

## Acknowledgments

I want to thank my wife
I want to thank my parents
I want to thank my advisor, Edward A. Lee
I want to thank all that worked on the PRET project with me:
Ben Lickly
Hiren Patel
Jan Rieneke
Sungjun Kim
David Broman
I would also like to thank the ptolemy group especially Christopher and Mary, Jia for providing me the template
I want to thank the committee members
I would like to thank everyone else that made this possible

# Contents

**List of Figures** **vi**

**List of Tables** **viii**

**1 Introduction** **1**
1.1 Motivation . . . 1
1.1.1 Timing Predictable Systems . . . 3
Timing Composability . . . 3
Timing Predictability . . . 4
1.2 Contributions . . . 4
1.3 Background . . . 5
1.4 Precision Timed Machines . . . 7

**2 Precision Timed Machine** **9**
2.1 Pipelines . . . 9
2.1.1 Pipeline Hazards . . . 9
Data Hazards . . . 9
Control Hazards . . . 11
Structural Hazards . . . 14
2.1.2 Pipeline Multithreading . . . 14
2.1.3 A Predictable Thread-Interleaved Pipeline . . . 16
Control Hazards . . . 17
Data Hazards . . . 18
Structural Hazards . . . 19
Deterministic Execution . . . 20
2.2 Memory System . . . 21
2.2.1 Memory Hierarchy . . . 22
Caches . . . 22
Scratchpads . . . 24
2.2.2 DRAM Memory Controller . . . 25
DRAM Basics . . . 25
Predictable DRAM Controller . . . 27
2.3 Instruction Set Architecture Extensions . . . 31
2.3.1 Timing Instructions . . . 33

Get_Time . . . 34
Delay_Until . . . 34
Exception_on_Expire and Deactivate_Exception . . . 35
2.3.2 Example Usage . . . 36
Constructing Different Timing Behaviors . . . 36
Timed Loops . . . 39

**3 Precision Timed ARM** **42**
3.1 Thread-Interleaved Pipeline . . . 43
3.2 Memory Hierarchy . . . 45
3.2.1 Boot code . . . 46
3.2.2 Scratchpads . . . 46
3.2.3 DRAM . . . 46
3.2.4 Memory Mapped I/O . . . 48
3.3 Exceptions . . . 48
3.4 Instruction Details . . . 50
3.4.1 Data-Processing . . . 51
3.4.2 Branch . . . 52
3.4.3 Memory Instructions . . . 52
Load/Store Register . . . 53
Load/Store Multiple . . . 54
Load to PC . . . 55
3.4.4 Timing Instructions . . . 56
Get_Time . . . 57
Delay_Until . . . 58
Exception_on_Expire and Deactivate_Exception . . . 59
3.5 Implementations . . . 60
3.5.1 PTARM VHDL Soft Core . . . 60
3.5.2 PTARM Simulator . . . 60
3.6 Timing Analysis . . . 61
3.6.1 Memory instructions . . . 62
3.6.2 Timing instructions . . . 63
3.6.3 Timed Loop revisited . . . 64
Obtaining the offset . . . 65
Overhead of the self compensating timed loop . . . 67
First loop iteration jitter . . . 67
3.6.4 Exceptions . . . 69

**4 Applications** **72**
4.1 Real-Time 1D Computational Fluid Dynamics Simulator . . . 72
4.1.1 Background . . . 73
4.1.2 Implementation . . . 75
Hardware Architecture . . . 75
Software Architecture . . . 78
4.1.3 Experimental Results and Discussion . . . 79

Timing Requirements Validation . . . 80
Resource Utilization . . . 81
4.1.4 Conclusion . . . 83
4.2 Eliminating Timing Side-Channel-Attacks . . . 83
4.2.1 Background . . . 84
4.2.2 A Precision Timed Architecture for Embedded Security . . . 85
Controlling Execution Time in Software . . . 86
Predictable Architecture . . . 87
4.2.3 Case Studies . . . 89
RSA Vulnerability . . . 90
An Improved Technique of using Deadline Instructions . . . 91
Digital Signature Algorithm . . . 92
4.2.4 Conclusion and Future Work . . . 93

**5 Related Work** **95**
5.1 Pipeline-Focused Techniques . . . 95
5.1.1 Static Branch Predictors . . . 95
5.1.2 Superscalar Pipelines . . . 96
5.1.3 VLIW architectures . . . 96
5.1.4 Multithreaded Pipelines . . . 97
Thread Scheduling . . . 97
Simultaneous Multithreaded Architectures . . . 98
5.1.5 Others . . . 99
Virtual Simple Architecture . . . 99
Java Optimized Processor . . . 99
MCGREP . . . 99
5.2 Memory-Focused Techniques . . . 100
5.2.1 Caches . . . 100
5.2.2 Scratchpads . . . 101
5.2.3 DRAM . . . 101

**6 Conclusion and Future work** **103**
6.1 Summary of Results . . . 103
6.2 Publications . . . 103
6.3 Future Work . . . 103

**Bibliography** **104**

# List of Figures

1.1 Program Execution Times [118] . . . 6
1.2 Simple Loop Timing Issues . . . 6
1.3 Timing anomaly cause by speculation [90] . . . 7

2.1 Sample code with data dependencies . . . 9
2.2 Handling of data dependencies in single threaded pipelines . . . 10
2.3 GCD with conditional branches . . . 11
2.4 Handling of conditional branches in single threaded pipelines . . . 12
2.5 Simple Multithreaded Pipeline . . . 15
2.6 Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages . . . 16
2.7 Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive . . . 18
2.8 Isolated execution of threads with a thread-interleaved pipeline . . . 21
2.9 Memory Hierarchy w/ Caches . . . 22
2.10 Memory Hierarchy w/ Scratchpads . . . 24
2.11 A dual-ranked dual in-line memory module. . . . 25
2.12 The periodic and pipelined access scheme employed by the backend [89]. . . . 28
2.13 Sketch of implementation of the backend [89]. . . . 29
2.14 Different Desired Timing Behaviors . . . 37
2.15 Timing diagram of different timed loops . . . 39

3.1 Block Level View of the PTARM 5 stage pipeline . . . 43
3.2 Four thread execution in PTARM . . . 45
3.3 Memory Layout of PTARM . . . 46
3.4 Example load by thread $i$ in the thread-interleaved pipeline. . . . 47
3.5 Integration of PTARM core with DMA units, PRET memory controller and dual-ranked DIMM [89]. . . . 47
3.6 Handling Exceptions in PTARM . . . 49
3.7 Data Processing Instruction Execution in the PTARM Pipeline . . . 51
3.8 Branch Instruction Execution in the PTARM Pipeline . . . 52
3.9 Load/Store Instruction Execution in the Ptarm Pipeline . . . 53
3.10 Load/Store Multiple Instruction Execution in the PTARM Pipeline . . . 55
3.11 Load to R15 Instruction Execution in the PTARM Pipeline . . . 56
3.12 Get_Time Instruction Execution in the PTARM Pipeline . . . 58

3.13 Delay_Until Instruction Execution in the PTARM Pipeline . . . . . 58
3.14 Implementation of Timer Unit . . . . . 59
3.15 PTARM Block Level View . . . . . 60
3.16 Timing details of get_time and delay_until . . . . . 63
3.17 Timing details of the *timer_expired* exception triggering . . . . . 64
3.18 Execution of the self compensating timed loop . . . . . 64
3.19 Jitter caused by initial timed loop setup . . . . . 67
3.20 Adjusted timed loop setup . . . . . 68

4.1 Design Flow . . . . . 74
4.2 High Level System Diagram . . . . . 74
4.3 Detailed System Diagram . . . . . 75
4.4 Library of Computational Node Elements . . . . . 75
4.5 The PTARM 6 Stage Pipeline . . . . . 76
4.6 System of PRET Cores and Interconnects . . . . . 77
4.7 Execution of Nodes at Each Time Step . . . . . 78
4.8 RSA Algorithm . . . . . 91
4.9 Run time distribution of 1000 randomly generated keys for RSA . . . . . 91
4.10 Digital Signature Standard Algorithm . . . . . 92

# List of Tables

2.1 Overview of DDR2-400 timing parameters of the Qimonda HYS64T64020EM-2.5-B2. [89] . . . . . 27
2.2 List of assembly timing instructions . . . . . 33

3.1 Exception vector table in PTARM . . . . . 49
3.2 List of assembly deadline instructions . . . . . 57
3.3 Timing properties of PTARM instructions (in thread cycles) . . . . . 61
3.4 Instruction execution trace of the self compensating timed loop (TC = thread cycles) 66
3.5 Exception_on_expire sample code timing details . . . . . 71

4.1 Table of supported pipe elements and their derived equations . . . . . 73
4.2 Computational Intensity of Supported Types . . . . . 80
4.3 Number of Occupied Slices per Core on the Virtex 6 (xc6vlx195t) FPGA. . . . . 81
4.4 Total Resource Utilization of Examples Synthesized on the Virtex 6 (xc6vlx195t) FPGA . . . . . 82

# Chapter 1

# Introduction

## 1.1 Motivation

*Cyber-Physical Systems* (CPS) are integrations of computation with physical processes [61]. In these systems, computation and physical process often form a tight feedback loop, affecting the behavior of each other. The embedded platforms and networks employed not only control the physical process, but at the same time monitor and adapt to the changes of the physical process. An enormous amount of applications can benefit from the potential of CPS. They include high confidence medical devices and systems, assisted living, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control, avionics, instrumentation, critical infrastructure control (electric power, water resources, and communications systems for example), distributed robotics (telepresence, telemedicine), defense systems, manufacturing, and smart structures. However, in order for CPS to be deployed in high confidence systems, such as advanced automotive or avionics systems, the platforms employed need to deal with two important properties of the physical process: they are inherently concurrent, and time progresses at its own pace.

Traditionally, real-time embedded systems have dealt with the notion of time. These systems impose deadlines and timing constraints to its underlying tasks to deliver services in real time. The timing constraints of *soft real-time systems* are typically used guarantee quality of service, while the constraints of *hard real-time systems* are used to guarantee safety critical tasks, so they must be met. The real-time embedded community has widely adopted techniques proposed for general purpose applications, believing that they will provide the same advantages and benefits for embedded systems. These include the programing language, the operating system, the tool-chains, and the computer architecture. However, these techniques are designed for general purpose systems that do not require stringent interaction with the physical environment. Thus, they emphasize on improving average performance over predictability. As a result, when computing systems absolutely must meet tight timing constraints, these recent computing advances often do more harm than good [60]. The scale and complexity of traditional embedded systems allowed designers to compensate with extra effort in design and analysis. However, these solutions begin to break down when transitioning to CPS.

In the current state of embedded software, nearly every abstraction has abstracted away *time*. The Instruction Set Architecture (ISA), meant to hide the hardware implementation details

from the software, does not include a timing semantic for the instruction executions. Widely adopted programming languages, meant to hide the details of the ISA from the program logic, do not express timing properties; timing is merely an accident of the implementation. Real-time operating systems (RTOS), meant to hide the details of the program from their concurrent orchestration, often use priorities to dictate the execution of tasks; the execution time of tasks can easily affect the scheduled outcome of execution. The lack of *time* in the abstraction layers lead to the following consequences:

- *Unnecessary complexities in the interaction of concurrent components* – This often is manifested when components share resources. For example, software threads are the typical abstractions for concurrent software written in C or Java. Because there is no guarantee of when a shared variable will be accessed by each thread, locks and semaphores are required to avoid race conditions. This not only introduces bugs, but also introduces complex and almost impossible to analyze interactions between threads [57]. As a result, there is great difficulty when synchronizing and communicating between components or tasks.

- *Unnecessary complexities in interactions across layers* – For example, scheduling could be done at multiple levels simultaneously without any coordination. As tasks or software threads are scheduled for execution in the OS, an explicit multithreaded dynamic dispatch architecture could also be scheduling instructions from different hardware threads without the knowledge of the OS [104].

- *Misleading or pessimistic analysis results when analyzing the whole system* – For example, task scheduling and context switching cost may vary from the cache or pipeline state change after executing each tasks. This is often not factored into the analysis [104]. Furthermore, because the large variation of execution time in modern complex processors, WCET analysis techniques often lead to overly conservative results for safety [118]. As the WCET is often the basis for priority of any scheduling scheme, the conservativeness is propagated throughout the system.

When the temporal properties of the system must be guaranteed, designers must reach beneath the abstraction layers, and understand thoroughly the complex underlying details and its affect on execution time. This not only increases the design complexity and effort, but the designed systems are *brittle* and extremely sensitive to change [92, 30]. For example, Sangiovanni-Vincentelli et al.[92] showed that when increasing the execution time of a task, any priority based scheduling scheme results in discontinuity in the timing of all tasks besides the task with the highest priority. At a lower level, adding a few instructions can easily result in a huge variation in program execution time; the state of the hardware dynamic prediction and speculation units, such as caches and pipelines, can easily be affected by the minimum program additions, causing misprediction penalties. Thus, in order to verify the timing of safety critical systems, the verification must be done on both the software system and its execution platform, they cannot be separated. This process is often time consuming and expensive. Since the abstraction layers do not give any temporal semantics to the system, each layer must be completely understood in order to reason and prove the timing properties of the full system. For avionics manufactures, this means stockpiling the same hardware for the lifetime of an aircraft; any upgrade of components or software in their system could result in drastic timing changes, and thus require re-certification.

### 1.1.1 Timing Predictable Systems

Thiele et al. [104], Henzinger [43] and Lee [60] have all identified the importance and difficulties of designing *timing-predictable systems*. Timing-predictable systems should exhibit the following property: *a small change in the input must not result in a large change in the output* [43]. If the definition of *output* includes the timing behavior exhibited by the system, then current abstracts disrupts this property at almost all levels.

A change is needed to efficiently and safely design next generation systems, especially if they effect the well being of our lives. In particular, how software and hardware deal with the notion of *time* needs to be more carefully understood and designed. At the lowest levels of abstraction, circuits and microarchitectures, timing is central to correctness. For example, in a microarchitecture, if the output of an ALU is latched at the wrong time, the ISA will not be correctly implemented. However, at higher levels, for example, the ISA, timing is hidden, and there is no temporal semantics; the execution time is irrelevant to correctness. Thus, each abstraction layer needs to be revisited to judiciously introduce some form of temporal semantics. Specifically for CPS, platforms must to be equip to handle the *inherent concurrency* and the *inexorable passage of time* for physical processes. Sangiovanni-Vincentelli et al. [92] identified these issues as the *timing composability* and *timing predictability* of systems, and lists them as requirements to enable efficient designs of large-scale safety-critical applications.

#### Timing Composability

Modern systems handle the concurrency of physical processes with multiple tasks, components or subsystems that are integrated together. In order to efficiently design the system, these individual parts are designed and tested separately, then later integrated to form the final system. This modularity of design is crucial for the continued scaling and improvement of systems. However, if component properties may be destroyed during integration, then the components can no longer be designed and verified separately. *Timing composability* refers to the ability to integrate components while preserving their temporal properties.

To preserve component properties during integration, modern designs often use a *federated architecture*. A Federated Architecture develops functions and features on physically separate platforms which are later integrated through an interconnect or system bus. As these features are only loosely coupled through an interconnect, interference is limited, allowing the preservation of certain properties independently verified. However, as each platform is feature specific, they are often idle during run time. In order to reduce resource consumption, there is a shift towards *integrated architectures* [77, 25], where multiple functions are integrated on a single, shared platform. Several challenges exists during this shift, but among them, it is crucial to guarantee that the timing properties are preserved during system integration. Only then, can designs continue to stay modular. Modern abstractions result in unnecessary complexity in the interaction of concurrent components, which leads to unpredictable interference between components. This hinders the ability to compose functions together on a shared resource while maintaining timing properties.

These challenges are present not only in research, but also in industry. The Integrated Modular Avionics (IMA) concept [86] aims to replace numerous separate processors and line replaceable units (LRU) with fewer, more centralized processing units in order to significant reduce the weight and maintenance savings in new generation of commercial airliners. AUTOSAR (AU-

Tomotive Open System ARchitecture)[1] is an architecture for automotive systems that is jointly being developed by manufacturers, suppliers and tool developers which attempts to defined standards and protocols to help modularize the design of these complex systems. We contend that in order for these standards to be safely defined, modern layers of abstractions that have been adopted from conventional computing advances must be redefined to allow for predictable composition of components.

**Timing Predictability**

In order to keep up with the continuous passage of time in physical processes, the system must be able to reason about its own passage of time. *Timing predictability* is the ability to predict timing properties of the system. Timing composition plays a big part of this when features are integrated, but even individually, it is difficult to analyze the execution time of programs.

Wilhelm et al. [118] described the abundant amount of research and effort that has been put into bounding the worst-case execution time(WCET). Not only is determining the worst case program flow a challenge, but the precision and usefulness of the analysis also depends on the underlying architecture[40]. Conventional architectures have proposed techniques that target the improvement of average case execution time (ACET) at the expense of execution time variability. As a result, it's extremely complex, if not impossible to obtain a precise bound of the execution time on modern architectures. The imprecision is often propagated through the system during integration, requiring pessimistic over-provisions to ensure timing requirements are met. Thus, time determinism and reduced jitter are needed for future systems to increase performance [92].

As modern layers of abstractions have no notion of *time*, the passage of time is a merely a consequence of the implementation. Therefore, existing techniques can only bound the WCET for a processor-program pair, and not the individual programs. Time bounds from the analysis are broken even when the underlying processor is upgraded to a newer model. Thus, the redefinition of abstraction layers must also include temporal semantics at different layers in order to reason about timing properties at higher levels.

## 1.2 Contributions

The contribution of this work is to propel design of cyber-physical systems from the lower levels of abstractions. Specifically in this thesis, we focus on the ISA abstraction layer, and its effects on microarchitecture design. The ISA defines the contract between software instructions and hardware implementations. Any correct implementation of an ISA will yield a consistent view of the processor state (eg. the contents registers or memory) for a given program developed with that ISA. However, modern ISAs do not specify timing properties of the instructions as part of the contract, and the benchmarks typically used to evaluate architectures compare them by the measured average performance. Thus, architecture designs have largely introduced techniques that improve average performance at the expense of execution time variability, leading to imprecise WCET bounds that limit the timing predictability and timing composability of CPS. The key challenges we contribute to are two fold.

First, we address the difficulty of predicting execution time and integrating multiple programs on modern computer architectures by proposing a new design paradigm for computer ar-

chitectures. Instead of performance, PREcision Timed (PRET) machines [30] are designed with *timing-predictability* as the main metric for success. We believe that as systems are becoming increasingly large and complex, increasing the speed of the underlying architecture through complexity will only do more harm than benefit. If we roll back computer architecture for 20 years, processors were all perfectly predictable. But the performance in terms of speed is undesirable and limiting, and we would be throwing away years of quality and technology enabling research. Thus, we do not intend to reinvent computing advancements, but instead evaluate them through the lenses of predictability and composability. In doing so, we gain a thorough understanding of the trade-offs between architectural performance gain and predictability, and show an architecture that is timing-predictable with reasonable performance.

Second, we address the lack of temporal semantics in the ISA by exploring instruction extensions that introduce timing semantics and control into programs. Introducing temporal semantics into any abstraction layer is a non-trivial task. Specifically for the ISA, over constraining the timing definitions of instructions could easily thwart architecture innovation opportunities. Instead we explore instruction extensions that aim to give temporal meaning to the *program*, not the individual instructions. These instruction extensions allow programmers to describe the passage of time within programs, and any architecture implementation of the extended ISA must abide to those descriptions. In doing so, we give temporal meaning to programs without limiting the innovation of architecture designs.

We contend that these two contributions must go hand in hand. Our ISA extensions provide the ability to give temporal meaning to programs, but do not enforce a predictable execution of its instructions. Without a predictable architecture, programs can still exhibit unpredictable execution time variances. On the other hand, a predictable architecture by itself does not bring temporal meaning to the programs, it merely executes them predictably. Time will still only be a side effect of the underlying implementation. However, with both the ISA extensions and the predictable architecture, we can equip platforms with the ability to handle physical processes and provide a solid foundation to propel the design of CPS at higher levels of abstractions.

## 1.3 Background

Programs manifest varying execution times. This is illustrated in figure 1.1, which shows the distribution of execution times exhibited by an arbitrary program on an arbitrary processor.

It highlights several key issues that are important to understanding program execution time. First, the *observable execution times* may not observe all *possible execution times*. This is important because far too often we rely on testing and end-to-end measurement to determine the WCET. This will, in general, overestimate the BCET and underestimate the WCET, and is not safe when timing must be guaranteed. Second, it is often difficult to determine the *actual* WCET, thus the worst case guarantee that is given is usually a bound on the WCET. The goal of the WCET analysis is to obtain a *safe* and *precise* bound on the WCET of a program [118]. *Safe* means that the execution time will never exceed the bounded time. *Precise* means that the bounded time is as close to the absolute WCET as possible.

Several factors contribute to the difficulties of a safe and precise WCET analysis. In general, it is impossible to obtain the upper bounds on execution times for programs because programs are not guaranteed to terminate. Real-time systems use a restricted form of programming to ensure

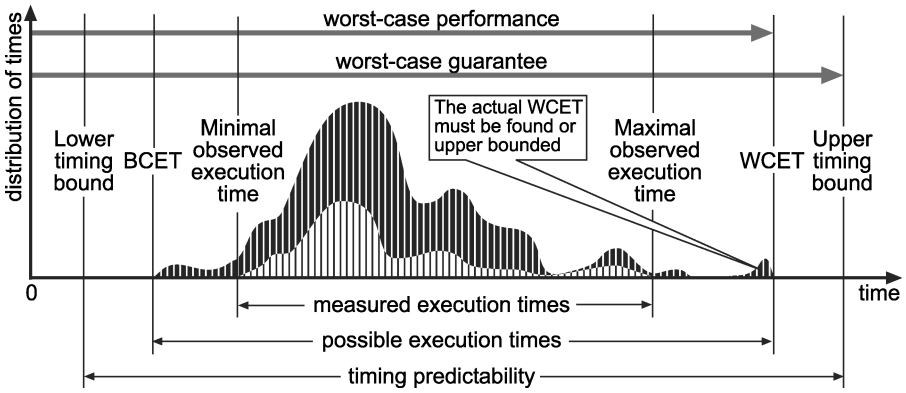



Figure 1.1: Program Execution Times [118]

an execution time upper bound. Recursion is often not allowed or must be explicitly bounded, as are the iteration counts of loops. Despite that, algorithms contain input dependent program paths that complicate analysis. The worst case program path depends on the worst-case input, which in general, is not known or hard to derive.

Along with complications from the software structure, the execution time variance exhibited by the underlying architecture further complicates analysis. A conventional microprocessor executes a sequence of instructions from an instruction set. Each instruction in the instruction set changes the state of the processor in a well-defined way. The microprocessor provides a strong guarantee about this behavior: a sequence of instructions *always* changes the processor state in the sequential order of the instructions. For speed, however, modern microprocessors rarely execute the instructions strictly in sequence. Instead, pipelines, caches, write buffers, and out-of-order execution reorder and overlap operations while preserving the illusion of sequential execution. This causes the execution time of even the same sequence of instructions to fluctuate, depending on the underlying execution of its instructions. To illustrate this, we show in figure 1.2 a code segment with a simple control structure and a static loop bound.

Even with a simple software structure, several situations can arise from the execution on the underlying architecture. Each array access in the code is compiled into a memory access. Whether the memory access hits or misses the cache has huge implications on program execution time. The *if* statement is usually compiled to a conditional branch, and the outcome of the branch predictor could easily affect the execution time of the program. Superscalar architectures can execute instructions out-of-order, thus data-dependencies in this code may or may not stall, depending on the memory accesses and how much loop unrolling is done by the compiler/architecture.

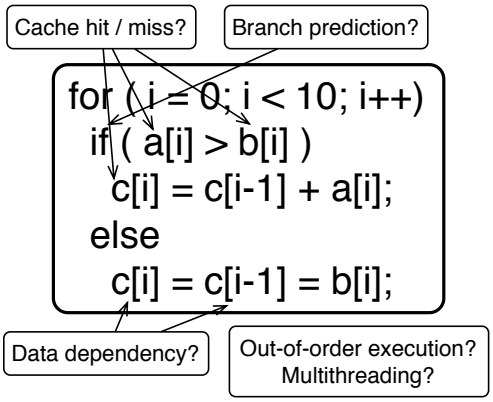



Figure 1.2: Simple Loop Timing Issues

Further complications arise as architectures become increasingly parallel with multiprocessing techniques such as multicore and multithreading. These techniques allow the architecture

to inherently handle concurrency, but can easily introduce temporal interference even between logically independent behaviors. For example, in a multicore machine with shared caches, the processes running on one core can affect the timing of processes on another core even when there is no communication between these processes. Similarly, Simultaneous Multithreading [105] architectures share a wide-issue superscalar pipeline across multiple hardware threads. Instructions are dispatched from all threads simultaneously using scoreboaring mechanisms. However, the contention for pipeline resources between threads can easily vary the execution time of a particular thread.

The common misconception is that at least a *safe* upper bound on the execution time can be easily determined by assuming the worse case in unknown situations. This is not true because dynamic processors can exhibit *timing anomalies* [90, 63], situations where a local worst-case does not entail the global worst-case. Reineke et al. [90] illustrates this with the example in figure 1.3. In this example, a mispredicted branch results in unnecessary instruction fetching that destroys the cache state. However, if the first instruction being fetched is a cache miss, the correct branch condition will be computed before the fetch, and no speculatively executed instruction will destroy the cache state. This example shows that simply assume a cache miss (local worst-case) will not always lead to the global worst-case execution time.

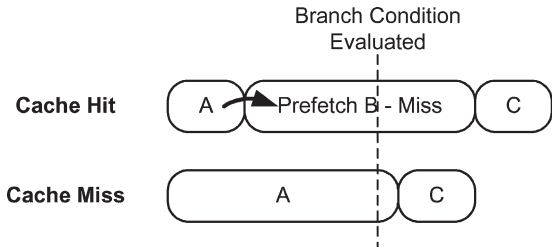


Figure 1.3: Timing anomaly cause by speculation [90]

The increasing complexity of architectures leads to the conclusion that the usefulness of the results of WCET analysis strongly depends on the architecture of the employed processor [40]. Modern processors employ features that improve average performance at the expense of worst-case performance, creating a large variation in execution time from the processor. These features are controlled and manage completely in hardware, not explicitly exposed to the software. As a result, decrypting the state of the processor to obtain reasonable execution time estimates is often extremely difficult, if not impossible, on modern architectures.

## 1.4 Precision Timed Machines

In this thesis we introduce the design and implementation of PREcision Timed (PRET) machines [30].

places timing predictability and composability as its first class citizen.

We aim to present architectures

With designs being pushed to higher and higher levels of abstraction, we need lower levels to provide robust, non brittle fundamentals in which we can reason about timing guarantees.

As Wilhelm et al. [119] quoted:

> *"The applicability of the AUTOSAR idea depends on availability of architectures on which software composition does not lead to unpredictable timing behavior."*

The remaining chapters are organized as follows. Chapter 2 explains the architecture of PRET including the thread-interleavedpipeline and memory hierarchy, Chapter 3, Chapter 4,

Chapter 5, Chapter 6.

# Chapter 2

# Precision Timed Machine

In this chapter we present the design principles of a PREcision Timed (PRET) Machine. Specifically, we discuss the implementation of a predictable pipeline and memory controller, and present timing extensions to the ISA. It is important to understand why and how current architectures fall short of timing predictability and repeatability. Thus, we first discuss the common architectural designs and their effects on execution time, and point out some key issues and trade-offs when designing architectures for predictable and repeatable timing.

## 2.1 Pipelines

The introduction of pipelining vastly improved the performance of processors. Pipelining increases the number of instructions that can be processed at one time by splitting up instruction execution into multiple steps. It allows for faster clock speeds, and improves instruction throughput compared to single cycle architectures. Ideally each in processor cycle, one instruction completes and leaves the pipeline as another enters and begins execution. In reality, different pipeline hazards occur which reduce the throughput and create stalls in the pipeline. The techniques introduced to mitigate the penalties of pipeline hazards greatly effect to the timing predictability and repeatability of architectures. We analyze several commonly used techniques to reduce the performance penalty from hazards, and show their effects on execution time and predictability.

### 2.1.1 Pipeline Hazards

#### Data Hazards

Data hazards occur when the data needed by an instruction is not yet available. Pipelines begin the execution of instructions before preceding ones are finished, so consecutive instructions that are data-dependent could simultaneously be executing in the pipeline. For example, the code in figure 2.1 shows assembly instructions from the ARM instruction set

```
add   r0, r1, r2        ; r0 = r1 + r2
sub   r1, r0, r1        ; r1 = r0 - r1
ldr    r2, [r1]         ; r2 = mem[r1]
sub   r0, r2, r1        ; r0 = r2 - r1
cmp  r0, r3             ; compare r0 and r3
```

Figure 2.1: Sample code with data dependencies

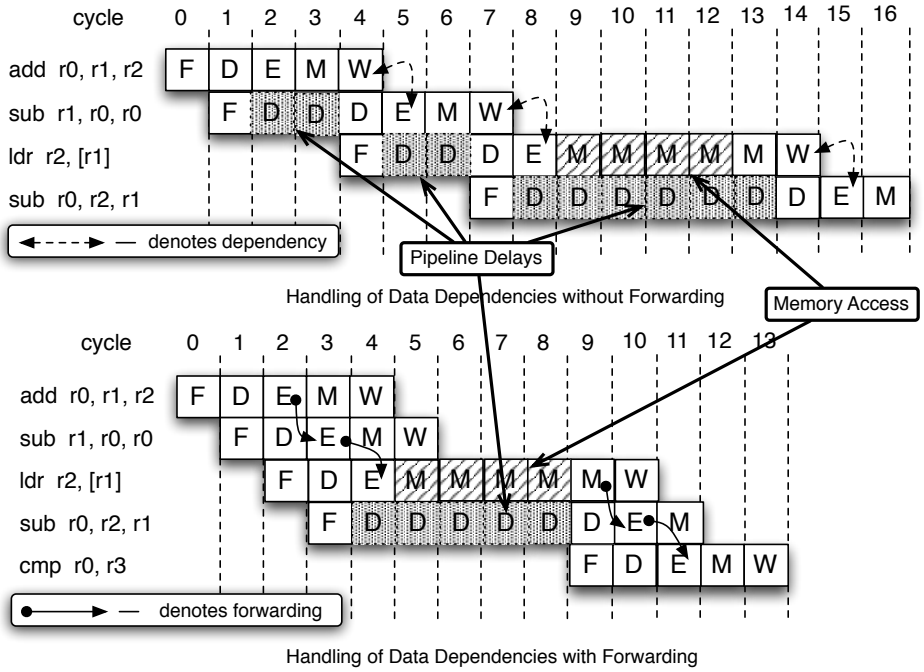



Figure 2.2: Handling of data dependencies in single threaded pipelines

architecture (ISA) that each depend on the result of its previous instruction. Figure 2.2 shows two ways data hazards are commonly handled in pipelines.

In the figure, time progresses horizontally towards the right. Each column represents a processor cycle. Each row represents an instruction that is fetched and executed within the pipeline. Each block represents the instruction entering the different stages of the pipeline – fetch (F), decode (D), execute (E), memory (M) and writeback (W). We assume a classic five stage RISC pipeline.

A simple but effective technique stalls the pipeline until the previous instruction completes. This is shown in the top of figure 2.2, as delays are inserted to wait for the results from previous instructions. The dependencies between instructions are explicitly shown in the figure to make clear why the pipeline delays are necessary. The performance penalty incurred in this case comes from the pipeline delays inserted.

*Data forwarding* is commonly used to mitigate the need for inserting delays when data hazards occur. Pipelines split up the execution of instructions into different execution stages. Thus, the results from an instruction could be ready, but waiting to be committed in the last stage of the pipeline. Data forwarding utilizes this and introduces backwards data paths in the pipeline, so earlier pipeline stages can access the data from instructions in later stages that have not yet committed. This greatly reduces the amount of delays inserted in the pipeline, as instructions can access the results of previous instructions before they commit. The circuitry of data forwarding usually consists of the backwards data paths and multiplexers in the earlier pipeline stages to select the correct data to be used. The pipeline controller dynamically detects whether a data-dependency exists, and changes the selection bits of the multiplexers accordingly to select the correct operands.

The bottom of figure 2.2 shows the execution sequence of the previous example in a pipeline with data forwarding. No pipeline delays are inserted for the first *sub* and *ldr* instruction because the data they depend on are forwarded with the forwarding paths. However, delays are

still inserted for the second *sub* instruction after the *ld* instruction. For longer latency operations, such as memory accesses, the results are not yet available to be forwarded by the forwarding paths, so pipeline delays are still required. This illustrates the limitations of data forwarding. They can address data hazards that result from pipelining, such as read-after-write register operations, but they cannot address data hazards that result from long latency operations, such as memory operations. More involved techniques such as the out-of-order execution or superscalars are required to mitigate the effects of long latency operations.

The handling of data hazards in pipelines can cause instructions to exhibit dynamic execution times. For example, figure 2.2 shows the *sub* instruction, in both top and bottom figures, exhibiting different execution times. To determine the execution time of instructions on pipelines that stall for data hazards, we need to determine when a stall is inserted, and how long the pipeline is stalled for. Stalls are required when the current instruction uses the results of a previous instruction that is still in execution in the pipeline. Thus, depending on the pipeline depth, a window of previous instructions needs to be checked to determine if any stalls are inserted. The length of the stall is determined by the execution time of the dependent instruction, because the pipeline will stall until that instruction completes. Data forwarding does not remove the data hazards, but only reduces the number of stalls required to take care of the data hazards. Thus, to determine the execution time when data forwarding is used, timing analysis needs to determine when the data forwarding circuitry cannot not forward the data for data hazards. This can similarly be done by observing the window of instructions not yet committed in the pipeline. The difference is, instead of all data dependencies, only data dependencies from long latency operations need to be detected.

Both techniques used for handling data hazards caused the execution time of instructions to depend on a window of previous instructions. The deeper the pipeline, the larger the window of instructions execution time will depend on. Thus, static execution time analysis needs to model and account for this additional the window of instructions on pipelined architectures that use stalling or forwarding to handle the data hazards.

### Control Hazards

Branches are the most common cause of control-flow hazards (or control hazards) in the pipeline; the instruction after the branch, which should be fetched the next cycle, is unknown until after the branch instruction is completed. Conditional branches further complicates matters, as whether or not the branch is taken depends on an additional condition that could also be unknown when the conditional branch is in execution. The code segment in figure 2.3 implements the *Greatest Common Divisor* (GCD) algorithm using the conditional branch instructions *beq* (branch equal) and *blt* (branch less than) in the ARM ISA. Conditional branch instructions in ARM branch based on conditional bits that are stored in a processor state register. Those conditional bits can be set based

```
gcd:
      cmp  r0, r1       ; compare r0 and r1
      beq  end          ; branch if r0 == r1
      blt  less         ; branch if r0 < r1
      sub  r0, r0, r1   ; r0 = r0 - r1
      b    gcd          ; branch to label gcd
less:
      sub  r1, r1, r0   ; r1 = r1 - r0
      b    gcd          ; branch to label gcd
end:
      add  r1, r1, r0   ; r1 = r1 + r0
      mov  r3, r1       ; r3 = r1
```

Figure 2.3: GCD with conditional branches

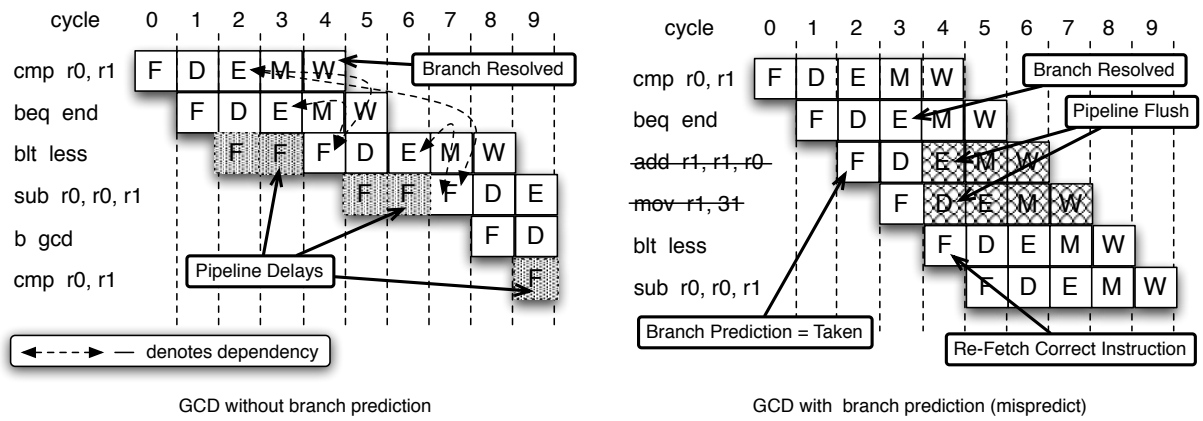



Figure 2.4: Handling of conditional branches in single threaded pipelines

on the results of standard arithmetic instructions [12]. The *cmp* instruction is one such instruction that subtracts two registers and sets the conditional bits according to the results. The GCD implementation shown in the code uses this mechanism to determine whether to continue or end the algorithm. Figure 2.4 shows the execution of the conditional branches from our example, and demonstrates two commonly used techniques to handling control hazards in pipelines. To show only the timing effects of handling control hazards, we assume an architecture with data forwarding that handles data hazards. As there are no long latency instructions in our example, all stalls observed in the figure are caused by the handling of control hazards.

Similar to data hazards, control hazards can also be handled by stalling the pipeline until the branch instruction completes. This is shown on the left of figure 2.4. Branch instructions typically calculate the target address in the execute stage, so two pipeline delays are inserted before the fetching of the *blt* instruction, to wait for *beq* to complete the target address calculation. The reasoning applies to the two pipeline delays inserted before the *sub* instruction. The performance penalty (often referred to as the *branch penalty*) incurred in this case is the two delays inserted after every branch instruction, to wait for the branch address calculation to complete.

To mitigate the branch penalty, some architectures enforce the compiler to insert one or more non-dependent instructions after each branch instruction. These instruction slots are called branch delay slots, and are always executed before the pipeline branches to the target address. This way, instead of wasting cycles to wait for the target address calculation, the pipeline continues to execute useful instructions before it branches. However, if the compiler cannot place useful instructions in the branch delay slot, *nops* need to be inserted into those slots to ensure correct program execution. Thus, branch delay slots are less effective for deeper pipelines because more delay slots need to be filled by the compiler to account for the branch penalty.

Instead of stalling, *branch predictors* are commonly employed to predict the branch condition and target address so the pipeline can speculatively continue execution. Branch predictors internally maintain some form of state machine that is used to determine the prediction of each branch. The internal state is updated after each branch according to the results of the branch. Different prediction schemes have been proposed, and some can even accurately predict branches up to 98.1% [66]. If the branch prediction is correct, no penalty is incurred for the branch because the correct instructions were speculatively executed. However, when the prediction is incorrect (often

referred as a *branch midpredict*), the speculatively executed instructions are flushed, and the correct instructions are re-fetched into the pipeline for execution.

The right of figure 2.4 shows the execution of GCD in the case of a branch misprediction. The *beq* branch is predicted to be taken, so the *add* and *mov* instructions from label *end* are directly fetched into execution. When *beq* progresses past the execute stage, *cmp* has forwarded its results used to determine the branch condition, and the branch target address has been calculated, so the branch is resolved. At this point, the misprediction is detected, so the *add* and *mov* instruction are flushed out of the pipeline. The next instruction from the correct path, the *blt* instruction, is immediately re-fetched, and execution continues. The performance penalty of branch mispredictions is derived from the number of pipeline stages between instruction fetch and branch resolution. In our example, the misprediction penalty is 2, as branches are resolved after the execute stage. This penalty only occurs on a branch mispredict, thus branch predictors with high success rates typically improve average performance of pipelines drastically, compared to architectures that simply stall for branches.

The two methods of handling control hazards exhibit vastly different effects on execution time. When stalls are used to handle control hazards, the execution time effects are static and predictable. The pipeline will simply *always* insert pipeline delays after a branch instruction. Thus, no extra complexity is added to the execution time analysis; the latency of branch instructions simply need to be adjusted to include the branch penalty. On the other hand, if a branch predictor is employed, the execution time of each branch will vary depending on the result of the branch prediction. To determine the success of a branch prediction, both the prediction and the branch outcome, both of which can dynamically change in run-time, must be known. Program path analysis can attempt to analyze the actual outcome of branches statically from the program code. The predictions made from the branch predictor depend on the internal state stored in the hardware unit. This internal state, updated by each branch instruction, must be explicitly modeled in order to estimate the prediction. If the predictor state is unknown, the miss penalty must conservatively be accounted for. There has been work on explicitly modeling branch predictors for execution time analysis [72], but the results only take into account the stalls from the branch penalty. Caches and other processor states are assumed to be perfect. In reality, the speculative execution on the predicted program paths lead to further complications that need to be accounted for. Other internal states exist in the architecture that could be affected by speculatively executing instructions. For example, if caches were used, their internal state could be updated during speculative execution of a mispredicted path. As architectures grow in complexity, the combined modeling of all hardware states in the architecture often lead to an infeasible explosion in state space for the analysis. This makes a tight static execution time analysis extremely difficult, if not impossible.

The difference in execution time effects between stalling and employing a branch predictor highlight an important trade-off for architecture designs. It is possible to improve average-case performance by making predictions, and speculatively executing based upon them. However, this comes at the cost of predictability, and a potential prolonging of the worst-case performance. For real-time and safety critical systems, the challenge remains to improve worst-case performance while maintaining predictability. How pipeline hazards are handled play an integral part of tackling this challenge.

Although less often mentioned, the presence of interrupts and exceptions in the pipeline also create control hazards. Exceptions can occur during the execution of any instruction, and

changes the control flow of the program to execute the exception handler. For single threaded pipelines, this means that all instructions fetched and not committed in the pipeline are speculative, because when an exception occurs, all uncommitted instructions in the pipeline become invalid. Pipelines often handle this by flushing all instructions and fetching the exception handler for execution. These effects are acknowledged, but often ignored in static analysis because it is simply impossible to model every possible exception, and its effect on the architecture states.

#### Structural Hazards

*Structural hazards* occur when a processor's hardware component is needed by two or more instructions at the same time. For example, a single memory unit accessed both in the fetch and memory stage results in a structural hazard. The design of the pipeline plays an integral part in eliminating structural hazards. For example, the classic RISC five stage pipeline only issues one instruction at a time, and uses separate instruction and data caches to avoid structural hazards. Structural hazards are generally much more prevalent in architectures that issue multiple instructions at a time. If structural hazards cannot be avoided, then the pipeline must stall to access the contended hardware component sequentially. The execution time effects of structural hazards are specific to how contention is managed for each pipeline design. Here we omit a general discussion of the timing effects, and later address them specifically for our proposed architecture.

### 2.1.2 Pipeline Multithreading

Discussed above, *data forwarding* and *branch prediction* are simple techniques employed to handle pipeline hazards. Advanced architectures, such as *superscalar* and *VLIW* machines, employ more complex mechanisms to improve the average performance of the architecture. Both architectures issue multiple instructions every cycle, and superscalar machines dynamically execute instructions out-of-order if no dependency is detected. These architectures exploit *instruction-level parallelism* to overlap the execution of instructions from a single thread whenever possible. On the contrary, *multithreaded architectures* exploit *thread-level parallelism* to overlap the execution of instructions from different hardware threads. Each hardware thread in a multithreaded architecture has its own physical copy of a processor state, such as the registers file and program counter etc. When a pipeline hazard arises from the execution of a hardware thread, another hardware thread can be fetched for execution to avoid stalling the pipeline. This improves the instruction throughput of the architecture.

Figure 2.5 shows the implementation of a simple multithreaded pipeline. It contains 5 hardware threads, so it has 5 copies of the Program Counter (PC) and Register files. The rest of the pipeline remains similar to a classic five stage RISC pipeline, with the addition of a few multiplexers used to select the thread states. Thus, the extra copies of the processor state and multiplexers are most of the hardware additions needed to implement hardware multithreading. When a hardware thread executes in the pipeline, its corresponding thread state is passed into the pipeline to be used. In most of this thesis, the term *threads* to refer to the explicit hardware threads that have hardware copies of the thread state. This is not to be confused with the common notion of *threads*, which describes software contexts managed by an operating system, with its states stored in memory. It will be explicitly noted when we refer to the software notion of threads. Ungerer et

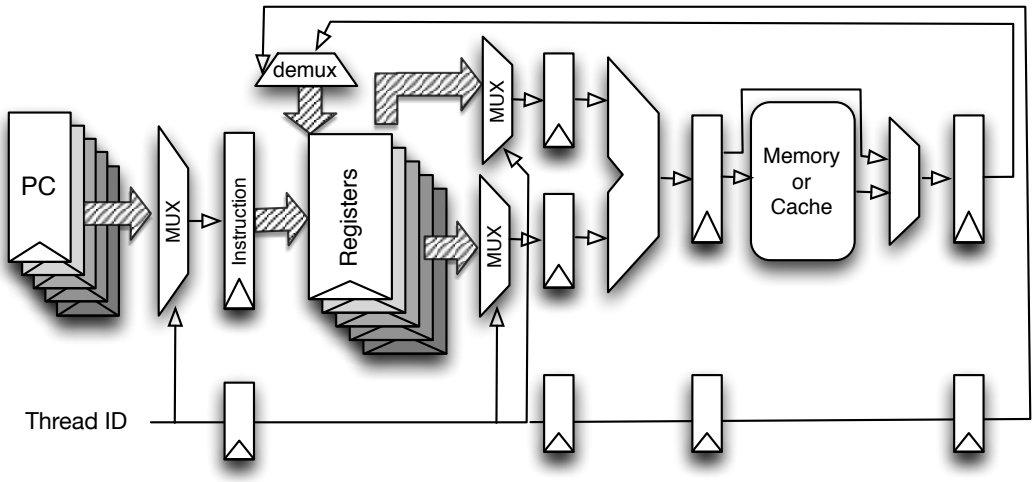



Figure 2.5: Simple Multithreaded Pipeline

al. [108] surveyed different multithreaded architectures and categorized them based upon the *thread scheduling* policy and the *execution width* of the pipeline.

The *thread scheduling* policy determines which threads are executing, and how often a context switch occurs. *Coarse-grain* policies manage threads similar to the way operation systems manage software threads. A thread gains access to the pipeline and continues to execute until a context switch is triggered. Context switches occur less frequently via this policy, so less threads are required to fully utilize the processor. Different coarse-grain policies trigger context switches with different events. Some policies trigger context switches on dynamic events, such as a cache miss or an interrupt; some policies trigger context switches on more static events, such as specialized instructions. *Fine-grain* policies switch context much more frequently – some as frequent as every processor cycle. The *execution width* of the pipeline is to the number of instructions fetched each cycle. Multithreaded architectures with wider pipeline widths can fetch all instructions a single thread, or mix instructions from different threads. The Sumultanous Multithreaded (SMT) architecture [105] is an example where instructions are fetched from different threads each cycle.

Multithreaded architectures present several challenges for static execution time analysis. As figure 2.5 illustrated, threads share the hardware components within the pipeline. If a hardware component, such as a branch predictor, maintains internal state, that internal state can be modified by all threads in the pipeline. As the internal states of the hardware components affect the execution time of the individual instructions, each thread can affect the execution time of all threads in the pipeline. If the threads' execution time are interdependent, their timing cannot be separately analyzed. As a result, in order to precisely model the hardware states, the execution order of instructions from all threads need to be known. The interleaving of threads depend heavily on the thread scheduling policy, execution width, and hazard handling logic employed in the pipeline. The compounding effect of these can create an overwhelming combination of possible thread interleavings, making static timing analysis nearly impossible, even if only a conservative estimation is desired.

Nonetheless, we contend that thread-level parallelism (TLP) *can* be exploited to handle pipeline hazards predictably. Even the most sophisticated architectures that fully exploit instruction-

level parallelism (ILP) cannot guarantee enough parallelism in a single instruction stream to remove all stalls caused by pipeline hazards. This is known as the *ILP Wall* [110]. Conventional multi-threaded architectures use coarse-grain thread scheduling policies to dynamically exploit TLP when there is not enough ILP to be exploited. However, the compounding effects of the combined architectural features lead to unpredictable architectural timing behaviors. Instead, a *thread-interleaved pipeline* fully exploits TLP with a fine-grained thread scheduling policy. We show that with several predictable architectural adjustments to the thread-interleaved pipeline, we can achieve a fully time-predictable pipeline with deterministic execution time behaviors.

### 2.1.3 A Predictable Thread-Interleaved Pipeline

Thread-interleaved pipelines use a fine-grain thread scheduling policy; every cycle a different hardware thread is fetched for execution. A round robin scheduling policy is often employed to reduce the context switch overhead every cycle. The thread-interleaved pipeline is known for implementing the peripheral processors of the Cray Multi-Threaded Architecture (MTA) multiprocessors [2]. Each the "peripheral processor" is implemented as a hardware thread. Interacting with input/output peripherals often lead to idle processor cycles that waits for the peripherals' responses. By interleaving several threads, thread-level parallelism is fully exploited, and the idle cycles can be used for simultaneous interaction with multiple input/output devices. Figure 2.6 shows an example execution sequence from a 5 stage single width thread-interleaved pipeline with 5 threads.

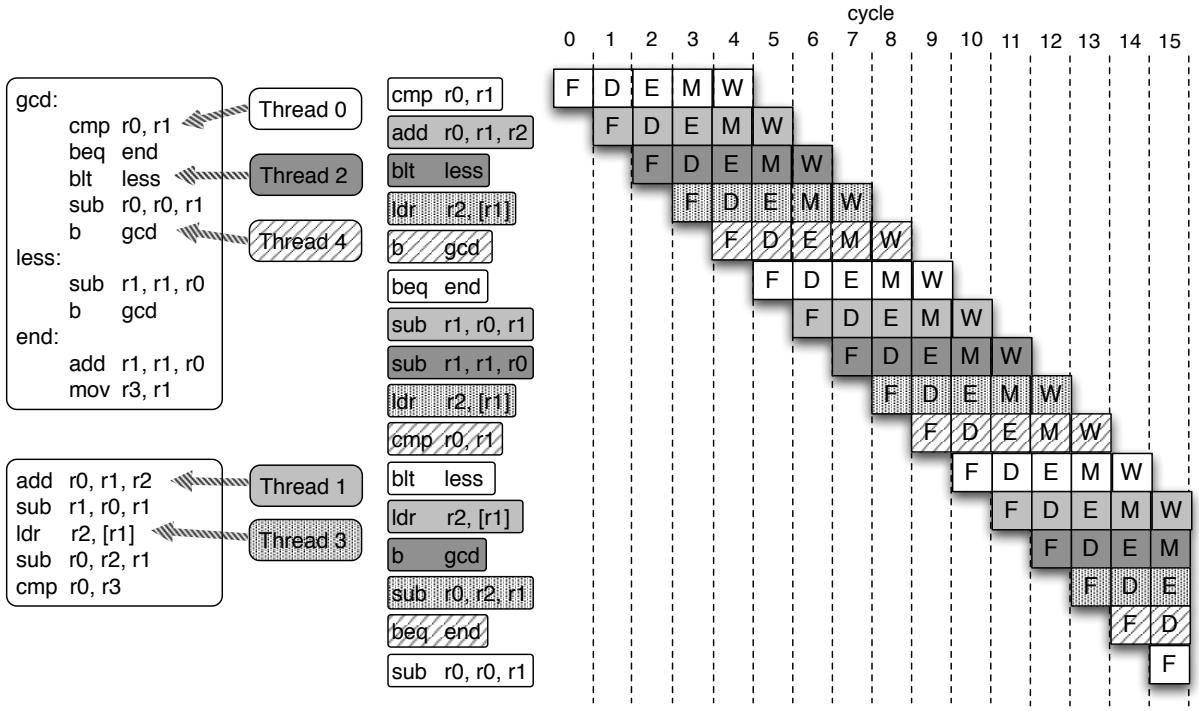



Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

The same code segments from figure 2.3 and figure 2.1 are used in this example. Threads 0, 2 and 4 execute GCD (figure 2.3) and threads 1 and 3 execute the data dependent code segment

(figure 2.1). The thick arrows on the left show the initial execution progresses of each thread at cycle 0. We observe from the figure that each cycle, an instruction from a different hardware thread is fetched in round robin order. By cycle 4, each pipeline stage is occupied by a different hardware thread. The fine-grained thread interleaving and round robin scheduling combine to form this important property for thread-interleaved pipelines, which provides the basis for a timing predictable pipeline design.

The interleaving of threads by itself does not guarantee timing predictability for the pipeline. Shared hardware components or a selective thread execution policy can easily allow the execution time of threads to be affected by each other. As previously discussed, a combined timing analysis of all threads in the pipeline is extremely difficult, if not impossible. In order for multithreaded architectures to achieve predictable performance, threads must be temporally isolated from one another. Temporal isolation removes cross-thread timing dependencies to allows timing analysis of threads independently. This enables a simple and more precise execution time analysis. We refine several features on the thread-interleaved pipeline to temporally isolate the threads and predictably handle pipeline hazards. This establishes a time-predictable thread-interleaved pipeline.

**Control Hazards**

By interleaving enough threads, control hazards can be completely removed in thread-interleaved pipelines. This can be observed from the execution sequence shown in figure 2.6.

At cycle 2, a *blt* instruction from thread 2 is fetched into the pipeline. In a single-threaded pipeline, a stall or branch prediction would be required before the next instruction fetch. However, as the figure illustrates, the next instruction fetched (*ldr*) at cycle 3 belongs to a different thread. There is no control hazard in this case, because the *ldr* instruction does not rely on the branch results of the *blt* instruction. Thus, no stall or branch prediction is needed to fetch this instruction. In fact, the branch result from *blt* is not needed until cycle 7, when thread 2 is fetched again. By this point, the branch has already been resolved, so no control hazard is caused from the *blt* instruction. The next fetched instruction from thread 2 is *always* from the correct program path. In this way, the control hazards from branches are eliminated.

The interleaving of threads also eliminate control hazards in the presence of exceptions. If the pipeline detects an exception for the *blt* instruction in its writeback stage (cycle 6), the control flow for thread 2 will be changed to handle the exception. Because no other instruction in the pipeline belongs to thread 2 at cycle 6, no instruction needs to be flushed. This reveals an important property our timing predictable pipeline, that *no instruction is speculatively executed*. The next instruction fetch from thread 2 does not occur until cycle 7. At that point, any control flow change, including one caused by an exception, is already known. Therefore, the correct program path is always executed.

The minimum number of threads required to eliminate control hazards depends on the number of the pipeline stages. Conservatively, interleaving the same number of threads as pipeline stages will always remove control hazards. Intuitively, this is because at any point in time, each stage of the pipeline with be executing an instruction from a different hardware thread. Thus, no explicit dependency will exist between instructions in the pipeline. Lee and Messerschmitt [58] further showed that it is possible to use one less thread than the number of pipeline stages for certain implementations. From here on, when we refer to the thread-interleaved pipeline, we assume enough threads to remove *explicit* dependencies between instructions in the pipeline.

Because control hazards are eliminated, branch predictors are not needed in our pipeline design. Removing the branch predictor contributes to the temporal isolation of threads, as the shared internal state of the branch predictor can create *implicit* dependencies between threads.

### Data Hazards

In a thread-interleaved pipeline, data hazards that stem from the pipeline of instructions are removed. The same reasoning for control hazard elimination is applied here, that no *explicit* dependencies exist between instructions in the pipeline, However, long latency operations can still cause data hazards in a thread-interleaved pipeline. This happens when a long latency operation is not completed before the next instruction fetch from the same thread. Although thread-interleaved pipelines can continue to fill the pipeline with other threads, if all threads simultaneously execute a long latency operation, then no thread will be available to fill the pipeline.

To maximize pipeline utilization and instruction throughput, thread-interleaved pipelines can mark threads inactive for long latency operations. However, this dynamic thread scheduling leads to non-trivial timing effects for the pipeline First, the number of active threads can fall below the minimum number of threads required to remove explicit dependencies of instructions in the pipeline. In this case, the eliminated control and data hazards are now reintroduced, and hazard handling logic, like the branch predictor, is required again. This can be circumvented by inserting pipeline stalls when the number active threads falls below the minimum. This is illustrated in figure 2.7. In the figure, 3 (out of 5) threads are interleaved through a 5 stage pipeline. We assume

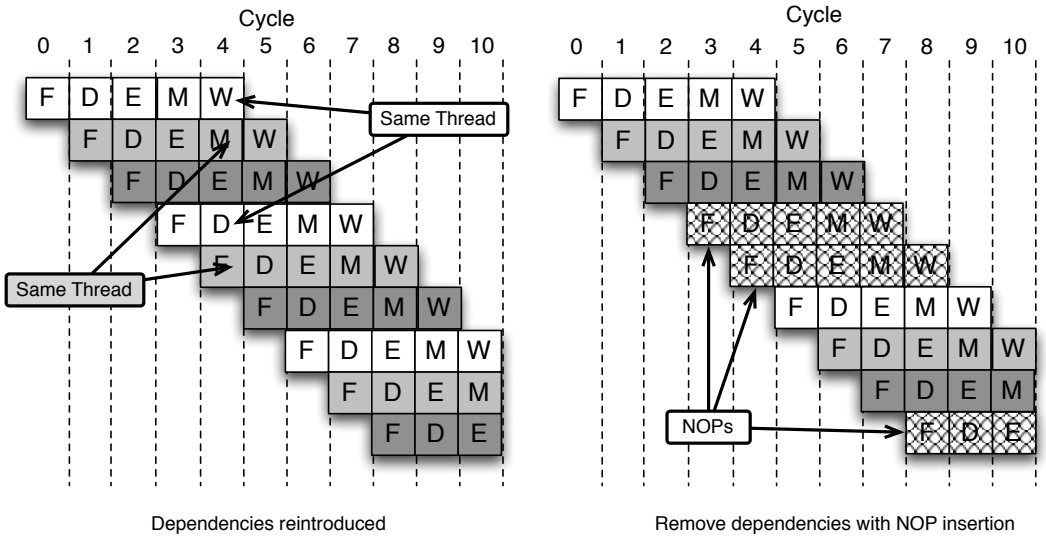



Figure 2.7: Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive

that the other 2 threads are inactive waiting for memory access. On the left we show that explicit dependencies between instructions in the pipeline are reintroduced. However, by inserting pipeline stalls to meet the minimum required thread count, the dependencies are once again removed. This is shown on the right. Employing more total threads in the pipeline can reduce the amount of stalling needed, since there is a larger pool of threads to select from. However, to ensure explicit dependencies are removed, stalls are *always* required when the active thread count drops below the

minimum.

More importantly however, the dynamic activation and deactivation of threads breaks temporal isolation between the threads. When a thread is deactivated, other threads are fetched more frequently into the pipeline. At any one moment, the execution frequency of threads would depend on the number of active threads. Because a thread can deactivate based upon its own execution and affect other threads' execution frequency, threads are no longer temporally isolated.

In order to maintain temporal isolation between the threads, threads cannot affect the execution time of others. For a time-predictable thread-interleaved pipeline, threads are not dynamically deactivated. Instead, when a thread is fetched in the presence of a data hazard, a pipeline delay is inserted to preserve the round robin thread schedule. This only slightly reduces the utilization of the pipeline, as other threads still executing during the long latency operation. But the temporal isolation of threads is preserved, as the execution frequency of threads remain the same regardless of any thread activity. Compared to single threaded pipelines, the benefits of latency hiding from mulithreading are still present.

Because no *explicit* dependency exists between the instructions in the pipeline, the forwarding logic used to handle data hazards can be stripped out in thread interleaved pipelines. Data forwarding logic contains no internal state, so threads are temporally isolated even if it is present. However, the pipeline datapath can be greatly simplified in the absence of forwarding logic and branch predictors. The static thread schedule reduces the overhead of context switches to almost none; it can be implemented with a simple $log(n)$ bit up-counter, where $n$ is the number of threads. This enables thread-interleaved pipelines to be clocked at faster clock speeds, because less logic exists between each pipeline stage.

### Structural Hazards

Threads on a multithreaded architecture, by definition, share the underlying pipeline datapath and any hardware unit implemented in it. Thus, multithreaded architectures are more susceptible to structural hazards, which can break temporal isolation if not handled predictably.

In multithreaded pipelines with a width of one, shared single-cycle hardware units do not cause structural hazards, because no contention arises from the pipelined instruction access. However, multi-cycle hardware units cause structural hazards when consecutive instructions access the same unit. The second instruction needs to wait for the first to complete before obtaining access. For thread-interleaved pipelines, this causes timing interference between threads, because consecutive instruction fetches come from different threads. One thread's access to a multi-cycle hardware unit can cause another to delay.

If it is possible to pipeline the multi-cycle hardware unit to be single-cycle accessible, the structural hazard and timing interference can be eliminated. In our time-predictable thread interleaved pipeline, floating point hardware units are pipelined to be single-cycle accessible. Hence, they are shared predictably between the hardware threads, and cause no timing interference.

If pipelining is not possible, then the management of contention for the hardware unit becomes essential to achieve temporal isolation of threads. The single memory unit in a thread-interleaved pipeline is an example of a shared, multi-cycle, non-pipeline-able hardware unit. In this situation, a time division multiplex access (TDMA) schedule can be enforced to remove timing interference. The TDMA schedule divides the access channel to the hardware unit into multiple time slots. Each thread only has access to the hardware unit at its assigned time slots, even if no

other thread is currently accessing the unit. By doing so, the access latency to the hardware unit is determined only by the timing offset between the thread and its access slot, not the activities of the other threads. In section 2.2 we show a predictable DRAM memory controller that use TDMA in the backend to schedule accesses to DRAM memory.

It is important to understand that a TDMA schedule removes timing interference, but does *not* remove structural hazards. In fact, a TDMA schedule can further uncover the performance penalty of structural hazards. By reserving privatized time slots for threads, the hardware unit will appear to be busy even when no thread is accessing it. Thus, structural hazards can occur even when the hardware unit is not being used. Although a TDMA schedule increases the average latency to access the hardware unit, the worst-case access latency is similar that of a conventional first-come-first-serve (FCFS) queuing based access schedule with a queue size of one. In both cases, the worst-case access latency is needs to account for the accesses of all threads. However, by using a TDMA schedule to predictably handle the structural hazards, the temporal isolation of threads enable a much tighter and simpler WCET analysis [64].

Even though shared single-cycle hardware units do not cause structural hazards, they can still introduce timing interference between threads in multithreaded architectures. Shared hardware units can create *implicit* dependencies between threads if the internal hardware states can be updated by any thread. A shared branch predictor, as discussed earlier, is a prime example for this. Our thread-interleaved pipeline removes the need for a branch predictor by the interleaving of hardware threads. A shared cache is another example. A cache maintains internal state that determines if a memory access goes to the cache or to main memory. There is typically an enormous latency difference between the two different accesses. When the cache is shared between threads, the different interleaving of threads can affect the execution time of all threads. It is even possible to degrade the performance of the system if threads continuously evict cache lines from each other. This phenomenon is known as *cache thrashing*. Partitioned caches [112] in this case can be used to enforce separate internal states, so each thread updates only its own internal state. Our time-predictable thread-interleaved pipeline employs scratchpads instead of caches. We discuss this in the context of a timing predictable memory hierarchy in section 2.2.

As a side note, the sharing of internal hardware states between threads also increases security risks in multithreaded architectures. Side-channel attacks on encryption algorithms [49] exploit the shared hardware states to disrupt and probe the execution time of threads running the encryption algorithm. The timing information can be used to crack the encryption key. We show in section 4.2 how our predictable thread-interleaved pipeline prevents timing side-channel attacks for encryption algorithms.

**Deterministic Execution**

The time-predictable thread-interleaved pipeline uses multithreading to improve instruction throughput, and maintains temporal isolation of threads to achieve deterministic execution. To highlight these features, we show the isolated execution of threads within a thread-interleaved pipeline. We use the example shown earlier (in figure 2.6), where we execute the sample GCD (figure 2.3) and data-dependent (figure 2.1) code on a 5 thread 5 stage thread-interleaved pipeline. Figure 2.8 shows the execution of the first two threads in isolation. Thread 0 executes GCD, and thread 1 executes the data-dependent code.

From the perspective of a thread, most instructions observe a 5 cycle latency, as shown in

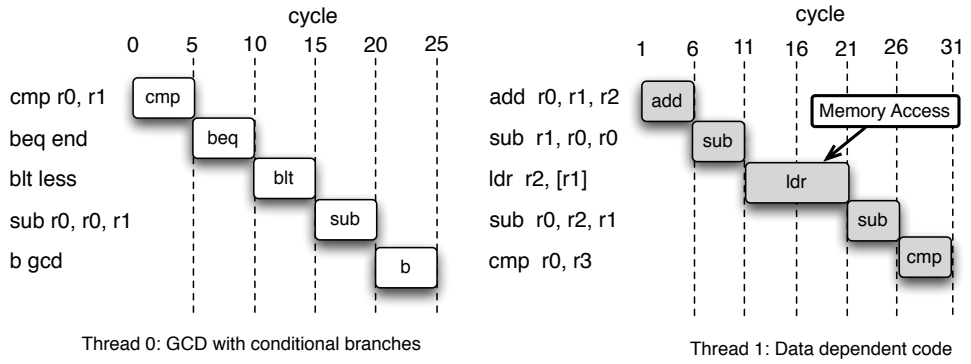



Figure 2.8: Isolated execution of threads with a thread-interleaved pipeline

figure 2.8. The minimum observable latency for instructions depend on the number of threads in the pipeline. This can also be understood as the latency for each thread between instruction fetches. In our time-predictable thread-interleaved pipeline, the static round robin thread schedule enables this latency to be constant. We use the term *thread cycle* to encapsulate this latency, and simplify the numbers for timing analysis. In our example, the instructions shown in thread 0 each take 1 thread cycle.

The *ldr* instruction in thread 1 accesses main memory. From the thread-interleaving, the access latency to main memory is hidden the concurrent execution of other threads. Thus, long latency instructions can appear to have a reduced latency in the isolated view of threads. In this example, the *ldr* instruction observes only a 2 thread cycle latency, even though the actual memory access latency could have been up to 10 processor cycles.

Threads are temporally isolated in our thread-interleaved pipeline, so execution of each thread can be analyzed in isolation. From the isolated view of each thread, each instruction completes its execution before the next one is fetched, and no instruction is executed speculatively. Because instructions do not overlap in execution, instruction's execution time is not affected by prior instructions. Control hazards are eliminated because a branch or exception is resolved before the next instruction fetch. The long latencies caused by structural or data hazards are hidden from the thread interleaving, improving the throughput of the pipeline. We will describe in detail our implementation of the thread-interleaved pipeline in the beginning of chapter 3.

## 2.2 Memory System

While pipelines designs continue to improve, memory technology has been struggling to keep up with the increase in clock speed and performance. Even though memory bandwidth can be improved with more bank parallelization, the memory latency remains the bottle neck to improve memory performance. Common memory technologies used in embedded systems contain a significant trade off between access latency and capacity. Static Random-Access Memories (SRAM) provide a shorter latency that allows single cycle memory access from the pipeline. However, the hardware cost to implement each memory cell prevents SRAM blocks to be implemented with high capacity. On the other hand, Dynamic Random-Access Memories (DRAM) uses a more compact memory cell design that can easily be combined into larger capacity memory blocks. But the mem-

ory cell of DRAMs must be constantly refreshed due to charge leakage, and the large capacity often prohibits faster access latencies. To bridge the latency gap between the pipeline and memory, smaller memories are placed in between the pipeline and larger memories to act as a buffer, forming a memory hierarchy. The smaller memories give faster access latencies at the cost of lower capacity, while larger memories make up for that with larger capacity but slower access latencies. The goal is to speed up program performance by placing commonly accessed values closer to the pipeline, while less access values are placed farther away.

### 2.2.1 Memory Hierarchy

#### Caches

A *CPU Cache* (or cache) is commonly used in the memory hierarchy to manage the smaller fast access memory made of SRAMs. The cache manages of contents of the fast access memory in hardware by leveraging the spatial and temporal locality of data accesses. The main benefits of the cache is that it abstracts away the memory hierarchy from the programmer. When a cache is used, all memory accesses is routed through the cache. If the data from the memory access is on the cache, then a cache hit occurs, and the data is returned right away. However, if data is not on the cache, then a cache miss occurs, and the cache controller fetches the data from the larger memory and adjusts the memory contents on the cache. The replacement policy of the cache is used to determine which cache line, the unit of memory replacement on caches, to replace. A variety of cache replacement policies have been researched and used to optimize for different memory access patterns of applications. In fact, modern memory hierarchies often contain multiple layers of hierarchy to balance the trade-off between speed and capacity. A commonly used memory hierarchy is shown in figure 2.9. If the data value is not found in the L1 cache, then it is searched for in the L2 cache. If the L2 cache also misses, then the data is retrieved from main memory, and sent back to the CPU while the L1 and L2 cache updates its contents. Different replacement policies can be used at different levels of the memory hierarchy to optimize the hit rate or miss latency of the memory access.

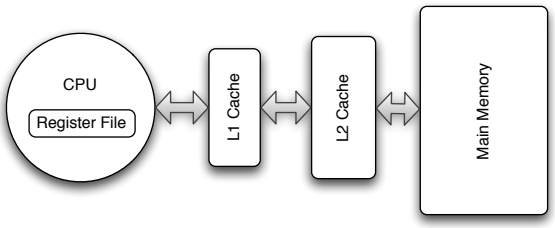


Figure 2.9: Memory Hierarchy w/ Caches

When caches are used, the program is oblivious to the different levels of memory hierarchy because it is abstracted away from the program; the cache gives its best-effort to optimize memory access latencies. Whether or not an access hits the cache or goes all the way out to main memory is hidden from the program. Thus, the programmer does not need to put in any effort, and can get a reasonable amount of performance. Furthermore, when programs are ported to another architecture with a different cache configuration, no change in the program is required to still obtain a reasonable amount of performance from the hardware. For general purpose applications, this gives the ability to improve design time and decrease design effort, which explains the cache's popularity.

However, the cache makes no guarantees on actual memory access latencies and program performance. The execution time of programs could highly vary depending on a number different factors – cold starts, previous execution contexts, interrupt routines, and even branch mispredictions that cause unnecessary cache line replacements. Thus, when execution time is important, the

variability and uncontrollability of caches may outweigh the benefits it provides.

The cache's internal states include the controller state and memory contents. As the programmer cannot explicitly control the state of the cache, it is extremely difficult to analyze execution time on systems caches. At an arbitrary point of execution, if the state of the cache is unknown, a conservative worst-case execution time analysis needs to assume the worst case, as if the memory access went directly to main memory. In order to acquire tighter execution time analysis, the cache must be modeled along program execution to predict the current cache state. The ease of such modeling depends on the replacement policy used in the cache.

For example, the *Least Recent Used* (LRU) replacement policy replaces the least recently used cache line whenever an eviction occurs. Within a basic block, a code segment without a control flow change, the contents of a cache with $N$ cache lines can be fully known after $N$ different memory accesses [40]. The $N$ different memory accesses will evict all cache lines in the cache prior to the basic block, and fill it with the memory contents of the $N$ accesses. In this case, the analysis assumes $N$ initial cache misses before the cache state is known. However, the cache state is destroyed when analysis hits a control flow merge with another path. Thus, the usefulness of this analysis depends on $N$ and how long basic blocks are in programs. In practice, the complexity of modern programs and memory architectures often introduce a high variability in program execution time, rendering analysis imprecise.

Even outside of the context of real-time applications, caches can present side effects that they were not intended for. For applications that require extreme high speed, the best-effort memory management that caches offer simply is not good enough. Programs often need to be hand tuned and tailored to specific cache architectures and parameters to acquire the desired performance. In order to tune algorithm performance, algorithm designers are required to understand to the abstracted away memory architecture and enforce data access patterns that conform to the cache size and replace policy. For example, instead of operating on entire rows or columns of an array, algorithms are rewritten to operate on a subset of the data at a time, or blocks, so the faster memory in the hierarchy can be reused. This technique is called *Blocking* [56], and is very well-known and commonly used.

Multithreaded threaded architectures with shared caches amongst the hardware threads can suffer from *cache thrashing*, an effect where different threads' memory accesses evict the cached lines of others. With multiple hardware threads, it is extremely difficult for threads have any knowledge on the state of the cache, because it is simultaneously being modified by other threads in the system. As a result, the hardware threads have no control over which level in the memory hierarchy they are accessing, and the performance highly varies depending on what is running on other hardware threads.

For multicore architectures, caches create a data coherency problem when data needs to be consistent between the multiple cores. When the multiple cores are sharing memory, each core's private cache may cache the same memory address. If one core writes to a memory location that is cached in its private cache, then the other core's cache would contain stale data. Various methods such as bus snooping or implementing a directory protocol [100] have been proposed to keep the data consistent in all caches. Implementing a scalable and efficient cache coherence scheme is still a hot topic of research today.

### Scratchpads

We cannot argue against the need for a memory hierarchy, as there is an undeniable gap between processor and DRAM latency. However, instead of abstracting away the memory hierarchy, we propose to *expose* the memory layout to the software.

*Scratchpads* were initially proposed for their power saving benefits over caches [14]. Scratchpads can be found in the Cell processor [37], which is used in Sony PlayStation 3 consoles, and NVIDIA's 8800 GPU, which provide 16KB of SPM per thread-bundle [76]. Scratchpads uses the same memory technology (SRAMs) as caches, but does not implement the hardware controller to manage its memory contents. Without the hardware controller, scratchpads do not manage its memory contents in hardware. Instead, scratchpads occupy a distinct address space in memory when they are used as fast access memory. Memory accesses that access the specific scratchpad address space will go to the scratchpad, and other accesses will go to main memory. Because in hardware scratchpads do not need to check whether the data is on the scratchpad or not, they have a reduced access latency, area and power consumption compared to caches [14].

Unlike caches, which overlay its address space with main memory to hide the hierarchy, scratchpads explicitly *exposes* the memory hierarchy, as figure 2.10 illustrates. The exposed memory hierarchy gives software full control over the management of memory contents in the hierarchy. Data allocated on the scratchpad will have single cycle access latencies, while other data will take the full DRAM access latency. The memory access latency for each request now depends only on the access address, and not that state of another hardware controller. This drastically improves the predictability of memory access times, and reduces the variability of execution time introduced with caches. However, this places the burden of memory management on the programmer.

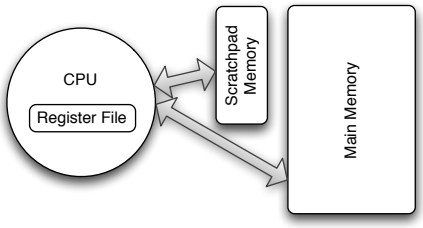


Figure 2.10: Memory Hierarchy w/ Scratchpads

Two allocation schemes are commonly employed to manage the contents of scratchpads in software. *Static allocation schemes* allocate data on the scratchpad during compile time, and the contents allocated on the scratchpad does not change throughout program execution. Static scratchpad allocation schemes [101, 80] often use heuristics or a compiler-based static analysis of the program to find the most commonly executed instructions or data structures, and allocate them statically on the scratchpad to improve program performance. *Dynamic allocation schemes* modify the data on the scratchpad during run time in software through DMA mechanisms. The allocation could either be automatically generated and inserted by the compiler, or explicitly specified by the user programmatically.

Embedded system designs often deal with limited resources and other design constraints, such as limited memory or hard timing deadlines. Thus, the design of such systems often require analysis on memory usage etc to ensure that the constraints are met. Higher level models of computations, such as Synchronous Dataflow (SDF) [59] or Giotto [42], allow users to design systems at a higher level. The higher level models expose the structure and semantics of the model for better analysis, which can be used to optimize scratchpad allocation dynamically. Bandyopadhyay [15] presented an automated memory allocation of scratchpads for the execution of Heterochronous Dataflow models. The Heterochronous Dataflow (HDF) model is an extension to the

Synchronous Dataflow (SDF) model with finite state machines (FCM). The HDF models contain different program states, each state executing a SDF model that contains actors communicating with each other. Bandyopadhyay analyzed the actor code and the data that was being communicated in each HDF state. The dynamic scratchpad allocation is inserted during state transitions, and the memory allocated is optimized for each HDF state. This allocation not only showed roughly 17% performance improvement compared to executions using LRU caches, but also more predictable program performance.

The underlying memory technology that is used to make both scratchpads and caches is not inherently unpredictable, as SRAMs provide constant low-latency access time. However, caches manage the contents of the SRAM in hardware. By using caches in the memory hierarchy, the hierarchy is hidden from the programmer, and hardware managed memory contents creates highly variable execution times with unpredictable access latencies. Scratchpads on the other hand exposes the memory hierarchy to the programmer, allowing more predictable and repeatable memory access performances. Although the allocation of scratchpads requires more effort, but it also provides opportunity for high efficiency, as it can be tailored to specific applications. Thus, in our time-predictable architecture, scratchpads are employed as our fast-access memory.

### 2.2.2 DRAM Memory Controller

Because of its high capacity, DRAMs are often employed in modern embedded systems to cope with the increasing code and data sizes. However, bank conflicts and refreshes within the DRAM can cause memory accesses to stall, further increasing the memory latency. Modern memory controllers are designed to optimize average-case performance by queueing and reordering memory requests to improve throughput of memory requests. This results in unpredictable and varying access times along with an increased worst-case access time for each memory request. In this section we will present a DRAM memory controller that privatizes DRAM banks with scheduled memory refreshes to provide improved worst-case latency and predictable access time. The contributions from this section is research done jointly with several co-authors from Reineke et. al [89]. We do not claim sole credit for this work, and the summary is included in this thesis only for completeness purposes. We will first give some basic background on DRAM memories, then present the predictable DRAM controller designed.

#### DRAM Basics

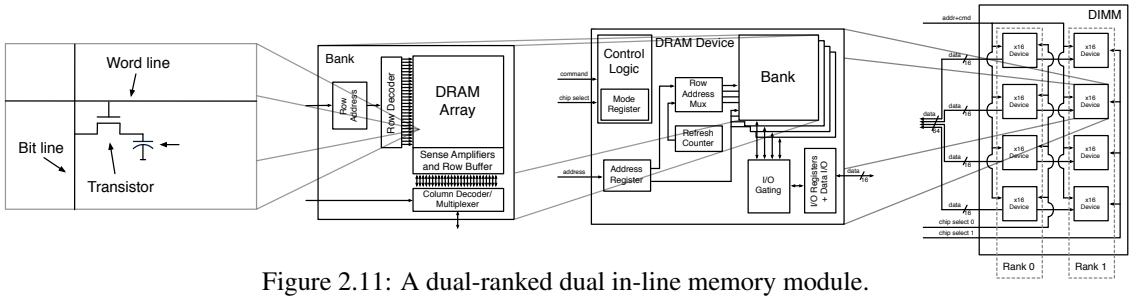



Figure 2.11: A dual-ranked dual in-line memory module.

Figure 2.11 shows the structure of a dual ranked in-line DDRII DRAM module. Starting from the left, a basic ***DRAM cell*** consists of a capacitor and a transistor. The capacitor charge

determines the value of the bit, which can be accessed by triggering the transistor. Because the capacitor leaks charge, it must be refreshed periodically, typically every 64 ms or less [47].

A ***DRAM array*** is made of a two-dimensional array of DRAM cells. Each access made to the DRAM array goes through two phases: a row access followed by one or more column accesses. During the row access, one of the rows in the DRAM array is moved into the row buffer. To read the value in the row buffer, the capacitance of the DRAM cells are compared to the wires connecting them with the row buffer. The wires need to be precharged close to the voltage threshold so the sense amplifiers can detect the bit value. Columns can be read and written to quickly after the row is in the row buffer.

The ***DRAM device*** consists of banks formed of DRAM arrays. Modern DRAM devices have multiple banks, control logic, and I/O mechanisms to read from and write to the data bus, as shown in the center of 2.11. Banks can be accessed concurrently, but the data, command and address busses, which is what the memory controller uses to send commands to the DRAM device, are shared within the device. The following table[1] lists the four most important commands and their function:

| Command | Abbr. | Description |
|---|---|---|
| Precharge | PRE | Stores back the contents of the row buffer into the DRAM array, and prepares the sense amplifiers for the next row access. |
| Row access | RAS | Moves a row from the DRAM array through the sense amplifiers into the row buffer. |
| Column access | CAS | Overwrites a column in the row buffer or reads a column from the row buffer. |
| Refresh | REF | Refreshes several[2] rows of the DRAM array. This uses the internal refresh counter to determine which rows to refresh. |

To perform reads or writes, the controller first sends the PRE command to precharge the bank containing the data. Then, a RAS is issued to select the row, and one or more CAS commands can be used to access the columns within the row. Accessing columns from the same row does not require additional PRE and RAS commands, thus higher throughput can be achieved by performing column accesses in burst lengths of four to eight words. Column accesses can immediately be followed by a PRE command to decrease latency when accessing different rows. This is known as auto-precharge (or closed-page policy). Refreshing of the cells can be done in two ways. First, by issuing a refresh command, which refreshes all banks of the device simultaneously. The refresh latency depends on the capacity of the device, but the DRAM device manages a counter to step through all the rows. The rows on the device could also be manually refreshed by performing row accesses to them. Thus, the memory controller could performance row accesses on every row within the 64 ms refresh period. This requires the memory controller to keep track of the refresh status of the device and issue more refresh commands, but each refresh takes less time because it is only a row access.

***DRAM modules*** are made of several DRAM devices integrated together for higher bandwidth and capacity. A high-level view of the dual-ranked dual in-line memory module (DIMM) is shown in the right side of figure 2.11. The DIMM has eight DRAM devices that are organized in two ranks. The two ranks share the address, command inputs, and the 64-bit data bus. The chip select is

[1] This table is as shown in [89]

[2] The number of rows depends on the capacity of the device.

used to determine which ranks is addressed. All devices within a rank are accessed simultaneously when the rank is addressed, and the results are combined to form the request response.

Our controller makes use of a feature from the DDR2 standard known as posted-CAS. Unlike DDR or other previous versions of DRAMs, DDR2 can delay the execution of CAS commands (posted-CAS) for a user-defined latency, known as the additive latency ($AL$). Posted-CAS can be used to resolve command bus contention by sending the posted-CAS earlier than the corresponding CAS needs to be executed.

Table 2.1 gives an overview of timing parameters for a DDR2-400 memory module. These timing constraints come from the internal structure of DRAM modules and DRAM cells. For example, $t_{RCD}, t_{RP}$, and $t_{RFC}$ are from the structure of DRAM banks that are accessed through sense amplifiers that need to be precharged. $t_{CL}, t_{WR}, t_{WTR}$, and $t_{WL}$ result from the structure of DRAM banks and DRAM devices. The four-bank activation window constraint $t_{FAW}$ constrains rapid activation of multiple banks which would result in too high a current draw. The memory controller must conform to these timing constraints when sending commands to the DDR2 module. Here we only gave a quick overview of DRAMs, we refer more interested readers to Jacob et al. [46] for more details.

| Parameter | Value [3] | Description |
|---|---|---|
| $t_{RCD}$ | 3 | **Row-to-Column delay**: time from row activation to first read or write to a column within that row. |
| $t_{CL}$ | 3 | **Column latency**: time between a column access command and the start of data being returned. |
| $t_{WL}$ | $t_{CL} - 1 = 2$ | **Write latency**: time after write command until first data is available on the bus. |
| $t_{WR}$ | 3 | **Write recovery time**: time between the end of a write data burst and the start of a precharge command. |
| $t_{WTR}$ | 2 | **Write to read time**: time between the end of a write data burst and the start of a column-read command. |
| $t_{RP}$ | 3 | **Row precharge time:** time to precharge the DRAM array before next row activation. |
| $t_{RFC}$ | 21 | **Refresh cycle time**: time interval between a refresh command and a row activation. |
| $t_{FAW}$ | 10 | **Four-bank activation window**: interval in which maximally four banks may be activated. |
| $t_{AL}$ | set by user | **Additive latency**: determines how long posted column accesses are delayed. |

Table 2.1: Overview of DDR2-400 timing parameters of the Qimonda HYS64T64020EM-2.5-B2. [89]

### Predictable DRAM Controller

We will split the discussion of the predictable DRAM controller into its backend and frontend. The backend translates memory requests into DRAM commands that are sent to the DRAM module. The frontend manages the interface to the pipeline along with the responsibility of scheduling the refreshes. Here we specifically refer to a DDR2 667MHz/PC2-5300 memory

[3] In cycles at 200 MHz

module operating at 200Mhz, which has a total size of 512MB over two ranks with four banks on each rank. While our discussion of the design of this DRAM controller is specific to our DDR2 memory module, the key design features are applicable to other modern memory modules.

**Backend** Conventional DRAM memory controllers view the entire memory device as one resource, and any memory requests can access the whole DRAM device. Subsequent memory accesses can target the same bank within the DRAM, which results in the need for memory requests to be queued and serviced sequentially, without exploiting bank parallelism. Our controller views the memory devices as independent resource partitioned by banks. Specifically, we partition our memory module into four *resources*, each consisting of two banks within the same rank. The banks within each resource can be arbitrarily chosen, but all banks within a resource must belong to the same rank, and each of the ranks must contain at least two resources. This is to avert access patterns that would incur high latency from the contention fo the shared busses within banks and ranks. The partitioning of the memory device allows us to fully exploit bank parallelism by accessing the resources in a periodic and pipelined fashion. The periodic access scheme to the four resources interleaves each memory access between the ranks. Subsequent accesses to the same rank goes to the other resource, grouped from banks. Figure 2.12 shows an example of the following access requests: read from resource 0 in rank 0, write to resource 1 in rank 1, and read from resource 2 in rank 0.

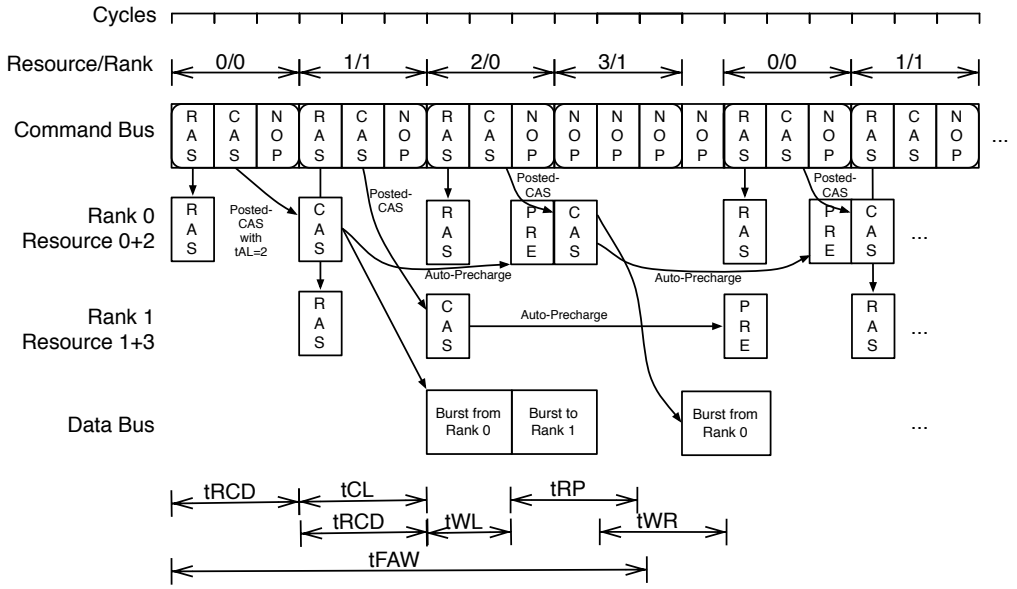



Figure 2.12: The periodic and pipelined access scheme employed by the backend [89].

Each access request is translated into a RAS (Row Access), posted-CAS (Column Access) and NOP command, which we call an access slot. The NOP command in the access slot is inserted between any two consecutive requests to avoid a collision on the data bus that occurs when a read request follows and a write request. This collision is cause by the one cycle offset between the read and write latencies. The RAS command moves a row into the row buffer, and the CAS command

accesses the columns within the row loaded into the row buffer. CAS commands can be either reads or writes, causing a burst transfer of $8 \cdot 4 = 32$ bytes that occupies the data bus for two cycles (as two transfers occur in every cycle). We send a posted-CAS instead of a normal CAS in order to meet the row to column latency shown in table 2.1. This latency specifies that the RAS command and the first CAS command need to be 3 cycles apart. However, figure 2.12 shows that manually issuing a CAS command to the first resource 3 cycles after its RAS command would cause a command bus conflict with the RAS command for the second resource. Thus, we instead set the additive latency $t_{AL}$ to 2 and use the posted-CAS that offsets the CAS command to conform to the row to column latency. This allows our memory controller to preserve our pipelined access scheme while meeting the latency requirements of the DRAM. We use a closed-page policy (also known as auto-precharge policy), which causes the accessed row to be immediately precharged after performing the column access (CAS), preparing it for the next row access. If there are no requests for a resource, the backend does not send any commands to the memory module, as is the case for resource 3 in 2.12.

Our memory design conforms to all the timing constraints listed in table 2.1. The write-to-read timing constraint $t_{WTR}$, incurred by the sharing of I/O gating within ranks, is satisfied by alternating accesses between ranks. The four-bank activation window constraint is satisfied because within any window of size $t_{FAW}$ we activate at most four banks within the periodic access scheme. Write requests with the closed-page policy requires 13 cycles to access the row, perform a burst access, and precharge the bank to prepare for the next row access. However, our periodic access scheme has a period of 12 cycles, as each access slot is 3 cycles, and there are four resources accessed. Thus, a NOP is inserted after the four access slots: to increase the distance between two access slots belonging to the same resource from 12 to 13 cycles. As a result, the controller periodically provides access to the four resources every 13 cycles. The backend does not issue any refresh commands to the memory module. Instead, it relies on the frontend to refresh the DRAM cells using regular row accesses.

A high level block view of our backend implementation is shown in figure 2.13. Each resource has a single request buffer and a respond buffer. These buffers are used to interface with the frontend. A request is made of an access type (read or write), a logical address, the data to be written for write requests. Requests are serviced at the granularity of bursts, i.e. 32 bytes in case of burst length 4 and 64 bytes in case of burst length 8. A modulo-13 counter is used to implement the 13 cycle periodic access scheme in our controller. The "resource" and "command" blocks are combinational circuits that are used to select the correct request buffer and generate the DRAM commands to be sent out. The "memory map" block is where logical addresses are mapped to physical addresses that determine the rank, bank, row and column to access. The data for read requests are latched into the response buffers to be read by the frontend.

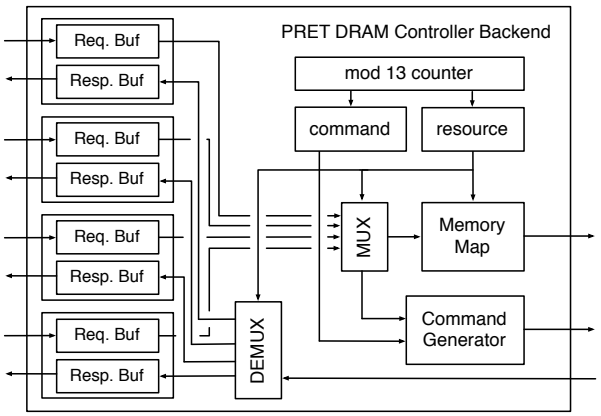



Figure 2.13: Sketch of implementation of the backend [89].

**Frontend** The frontend of our memory controller manages the interfacing to our backend, and the refreshing of the DRAM device. The privatization of DRAM banks creates four independent resources that is to be accessed separately from the front end. Thus, our memory controller is designed to be used by multicore or multithreaded architectures that contain multiple requesters which need access to the main memory. Several recent projects strive to develop predictable multi-core architectures, such as those proposed by the MERASA [107], PREDATOR [**?**], JOP [98], or CoMPSoC [39] projects, which require predictable and composable memory performance. These could potentially profit from using the proposed DRAM controller.

Specifically, we designed this memory controller to interface with the thread-interleaved pipeline discussed previously in section 2.1.3. The thread-interleaved pipeline contains multiple hardware threads that each require access to main memory. We assign each hardware thread to a private memory resource, and send out memory requests to the memory controller frontend, which receives the request and places it within the request buffer. Each thread in the thread-interleaved pipeline sends out only one outstanding memory request at a time, so the single request buffer for each resource is sufficient to interface with our thread-interleaved pipeline. Once the request is serviced from the backend, the pipeline can read the data from the response buffer, and prepare to send another memory request. In section 3.2 we will detail how our implemented thread-interleaved pipeline interfaces with this predictable DRAM controller, and discuss the memory access latency of this interaction.

**Shared Data** The privatization of resources for predictable access means that there is no shared data in the DRAM. This serves as an interesting design challenge, as it is impossible to assume no communication between contexts in a multicore or multithreaded environment. In our implementation, which we will detail in section 3.2, the scratchpads can be configured to be shared between the hardware threads for communication. This can be done because the scratchpad and DRAM memory has distinct address regions, so no shared memory space will overlap onto the DRAM address space. Most multi-core processors use DRAM to share data while local scratchpads or caches are private. In this case, the sharing of data on the DRAM can be achieved by arbitrating accesses in the frontend. The four resources in the backend can be combined into one, and any access to this single resource would result in four smaller accesses to all the backend resources. This single resource could then be shared among the different cores of a multi-core architecture using predictable arbitration mechanisms such as Round-Robin or CCSP [9] or predictable and composable ones like time-division multiple access (TDMA). This sharing of DRAM resources comes at a cost of increased memory access latency, which is detailed in [89].

**Refreshing the DRAM** The frontend of our memory controller also manages the refreshing of DRAM cells. DRAM cells need to be refreshed at least every 64 ms. Conventionally this is done by issuing a hardware refresh command that refreshes several rows of a device at once[4]. Hardware refresh commands have longer refresh latencies each time a refresh is issued, but requires less refresh commands to meet the refresh constraints posed by the DRAM. However, when the hardware refresh command is issued, all banks in the target DRAM device are refreshed, prohibiting any other memory access to the device. In our backend, this would extend across multiple resources, causing multiple resources to be blocked for memory accesses. Memory access latencies

[4]Internally, this still results in several consecutive row accesses.

now need to account for potential refresh command latencies, which varies depending on the refresh progress. Instead, we use the distributed, RAS-only refresh [70] to each bank separately. Memory refreshes is in this case is equivalent to a row accesses to a bank, and each resource can be refreshed without effecting others. Manually accessing rows on the other have much shorter latencies each time, but incurs a slight bandwidth hit because more accesses need to be performed to meet the refresh constraints. The shorter latencies however improves the worst-case access latency, because the refresh latency is shorter.

When a refresh is required can be statically analyzed. In our device, each bank consists of 8192 rows, so each row has to be refreshed every $64ms/8192 = 7.8125\mu s$. At a clock rate of 200 MHz of the memory controller, this corresponds to $7.8125\mu s \cdot (200cycles/\mu s) = 1562.5$ cycles. Since each resource contains two banks, we need to perform two refreshes every $1562.5$ cycles, or one every $781.25$ cycles. One round of access is 13 cycles at burst length 4, and includes the access slots to each resource plus a nop command. So in the frontend we schedule a refresh every $\lfloor 781.25/13 \rfloor^{th} = 60^{th}$ round of the backend. If no memory access is in the request buffer for the resource being scheduled for refresh, then the row refresh can be directly be issued. Typically when a contention between a memory request and a refresh occurs, the refresh gets priority so the data can be retained in the DRAM cell. However, our refresh schedule schedules refreshes slightly more often than necessary. Scheduling a refresh every $60 \cdot 13$ cycles means that every row, and thus every DRAM cell, is refreshed every $60 \cdot 13$ *cycles* $\cdot\ 8192 \cdot 2/(200000$ *cycles*$/ms) \leq 63.90ms$. We can thus push back any of these refreshes individually by up to $0.1ms = 20000$ cycles without violating the refreshing requirement. So in our frontend, the memory request is serviced first (which takes 13 cycles), then the refresh is issued in the next access slot.

In section 3.2 when we detail the interaction between our thread-interleaved pipeline and the memory controller, we will show that the synchronization of the thread-interleaved pipeline to our controller backend allow us to completely hide memory refreshes in some unusable access slots lost in the synchronization. Thus, providing predictable access latencies for all load/store instructions to the DRAM through our DRAM controller.

## 2.3 Instruction Set Architecture Extensions

The Instruction-set architecture (ISA) serve as the contract between the software and the hardware. The programmer understands the semantics of each instruction and use it to construct programs. Computer architects ensure that the implementation of each instruction abides to the semantics specified in the ISA. The semantics of the instructions in modern ISAs often do not specify temporal meaning to the instructions. Thus, in order to reason about the temporal properties of a program, we must step outside of the ISA semantics and dive deep into the architectural details. Since ISAs do not provide any means of exposing or controlling the timing behavior of software, their implementations are under no obligations to exhibit predictable and repeatable timing behaviors. This makes the reasoning of temporal behaviors of programs even more difficult. In the previous sections, we presented a predictable computer architecture that implements timing predictable behaviors for conventional instructions in the ISA. In this section, we will present our initial efforts to extend the ISA with timing properties. Our vision is to bring temporal meaning to the semantics of ISA which allows us to reason about timing of programs independent of the platform. This allows higher-level models with temporal semantics, such as models expressed using e.g. MathWorks

Simulink® or Giotto [42], to be more easily synthesized into lower-level implementations, such as C code, without deeply coupling the design to a particular hardware platform.

A naive way to the extend the ISA with timing properties would be to associate with each instruction a constant execution time. This *constant time* ISA provides a clear timing definition to all programs written with it. The semantics of the program would include the execution time of basic blocks, and any underlying architecture implementation must conform to it. All programs written with the *constant time* ISA can also be ported across different architectures of the same family and maintain the same timing behavior. This also means that any architecture implementation that does not exhibit the defined timing properties is an incorrect implementation. A *constant time* ISA would allow the reasoning of temporal properties independent of architecture, and engrave in the semantics of programs temporal definitions. However, the major limitation of the *constant time* ISA is that it prevents performance improvements at the micro-architectural level, as instruction execution time must conform to the constant time specified in the ISA. Modern ISAs allow computer architects to freely innovate in architectural techniques to speed up execution time of instructions while still conforming to the semantics of the ISA. The program performance improves as the architecture performance improves, without any effort from the programmer. By associating a constant execution time for each instruction, the *constant time* ISA over constrains the performance of programs, and limits the innovation of architecture implementations.

Instead of associating with all instructions a constant execution time, we extend the ISA by adding assembly level instructions that allow us to control the timing behavior of programs. Ip and Edwards [45] proposed a simple extension to the processor which implemented the *deadline* instruction, an instruction that allows the programmer to specify a minimum execution time of code blocks. They showed an implementation of a VGA controller that uses the deadline instructions to send out the horizontal and vertical sync signals in software. We further expand on this concept of controlling execution time in software, and introduce a set of assembly timing instructions that allows us to control not only the minimum execution time, but also handle cases where the execution time exceeds a specified deadline.

It is currently already possible to manipulate external timers and set interrupts in most modern embedded platforms. However, the procedure of setting timing interrupts highly varies depending on platform implementation. Access to external timers are often done through memory mapped registers, which views the external timer as another I/O component. As a result, the timing behavior of the program is deeply tied to the underlying implementation platform. By defining the timing instructions as part of the instruction set, we unify the semantics of time across all programs implemented using the ISA, and any correct implementation must conform to the timing specifications in the software. This brings the *control* of timing up to software, instead of it being a side effect of the underlying architecture implementation. In this section, we will introduce the timing instructions added to the instruction set that allow us to experiment and investigate the effects and possibilities of extending ISA with timing properties. Formally defining the ISA extensions is part of an ongoing work for the PRET project. Here, we describe informally their semantics and through illustrative examples we will also present their usage. In section 3.4 we will present the implementation and timing details of these instructions.

### 2.3.1 Timing Instructions

Our extension of the ISA assumes a *platform clock* that is synchronous with the execution of instructions. This *platform clock* is used by all timing instructions to specify and manipulate the execution time of code blocks. The representation of *time* in the ISA is in itself an interesting topic of research. For example, IEEE 1588 timestamps use 32 bits of nanoseconds and 48 bits of seconds to represent time. Our current implementation uses 64 bits of nanoseconds in the *platform clock* to represent time. We choose this representation for several reasons. First, with our timing instructions, timestamps are obtained by the programmer and can be manipulated throughout the program with data operating instructions. Typical datapaths and registers are 32 bits. By using 64 bits of nanoseconds to represent time, programmers can use *add with carry* instructions to manage with overflow of 64 bit additions without extra overhead. On the other hand, if we used the IEEE 1588 timestamp format to represent time, then any manipulation of time through the software would require explicit checking of the nanoseconds overflowing to the seconds register. Second, the 64 bit nanoseconds simplifies the hardware implementation and comparisons of the *platform clock* and timestamp values. In chapter 3 we will show our implementation, which utilizes the existing datapath and integrates the *platform clock* deeply into the architecture.

Unsigned 64 bits of nanoseconds can only represent time up to a little more than 584 years, so the *platform clock* in our ISA is meant for a local representation of time. The *platform clock* is reset to zero on processor reset. Even though the timing instructions operate on the exact 64 bit value of time, they are used control offsets of time. The actual value of the timestamp is merely used to calculate the elapsed time of code blocks. For distributed systems that require communication of timestamps across platforms, the consistent view of time across platforms must be obtained. This can occur during system initialization, where the global time is obtained and kept in the system. This initial global time can be appended to the current platform time to obtain the current global time. For systems designed run longer than 584 years each reset, the overflow of the 65th bit must be managed in software to ensure a consistent view of time.

| Instruction | Description |
|---|---|
| ***get_time*** | *get_time* is used to obtain the current platform time. |
| ***delay_until*** | *delay_until* is used to delay the execution of the program until a certain timestamp. |
| ***exception_on_expired*** | *exception_on_expire* is used to register timestamps that trigger timing exceptions when the platform time exceeds the registered timestamp. |
| ***deactivate_exception*** | *deactivate_exception* is used to deactivate the registered timestamps that trigger timing exceptions. |

Table 2.2: List of assembly timing instructions

Table 2.2 shows the timing instructions and a brief description of their functionality. Our current implementation extends the ARM [12] instruction set, so here we will present our timing instruction extensions in the context of the ARM ISA. However, the concepts and extensions could easily be applied to other ISAs. The ARM ISA sets aside an instruction encoding space to allow additions to the architecture with co-processor extensions. Our timing instructions are currently implemented using the co-processor instruction encoding, which also enables us to use conventional ARM cross-compilers to compile programs and test our architecture.

### Get_Time

The *get_time* instruction is mainly used to obtain the time on the *platform clock*. This instruction interfaces the program with the current platform time by loading the 64 bit timestamp of the current platform time in general purpose registers. The timestamps are stored in general purpose registers to make it accessible to programmers. The programmer can manipulate the timestamps by using conventional data-processing instructions like add or subtract. However, because the timestamp is 64 bits, architectures with 32 bit registers store the value in 2 separate registers. Thus, any manipulation of timestamps must handle the overflow properly from 32 bits operations. Several ISAs provide an *add with carry* instruction that can be used, or else the programmer must explicitly do so in software. The timestamp is used as inputs to other timing instructions which we will introduce below.

If the *platform clock* was memory mapped to two 32 bit memory locations, then the functionality of this instruction could technically be implemented by the reading of memory mapped addresses. This would be similar to conventional methods of accessing timers. However, loading a 64 bit time value would possible require 2 consecutive loads. Without care, the programmer could easily read 2 inconsistent 32 bit values of time, because the platform time continues to elapse in between the 2 reads. Even if a 64 bit load instruction is present in the ISA, the ISA makes no guarantee that a loaded 64 bit value from main memory would contain a consistent timestamp value from the same point in time. Thus, to make explicit the nature of the operation, we use a separate instruction that ensures the programmer will get a consistent 64 bit timestamp from a single point in time. In our implementation, this single point of time is when the *get_time* instruction enters the pipeline.

### Delay_Until

The *delay_until* instruction is used to delay program execution until a specified time. The effect is similar to the one presented by Ip and Edwards [45], where the programmer can specify a minimum execution time for a code block. The difference is, in our ISA, the unit of time is represented by nanoseconds, instead of processor cycles. The *delay_until* instruction takes as input a timestamp, usually derived from the timestamp obtained from *get_time*, and compares it to the current platform time to determine if delays are needed. Listing 2.1 shows an example of how *delay_until* and *get_time* are used together to control a minimum execution time a code block. The assembly code is written using the ARM instruction set. The timing instructions are implemented as co-processor 13 instructions, so all timing instructions are in the format ***cdp, p13, <opcode> rd, rn, rm, 0***. *Get_time* has an opcode of 8, and *delay_until* has an opcode of 4.

Listing 2.1: Sample assembly code of delay_until

```
1 cdp p13, 8, c2, c0, c0, 0  ; {c2,c3} = platform time (get_time)
2 adds r3, r3, #400          ; c3 += 400 (save carry)
3 adc r2, r2, #0             ; c2 = c2 + <previous carry>
4
5 add r5, r6, r6             ; code block to execution
6
7 cdp p13, 4, c2, c2, c3, 0  ; delay_until
8 b end
```

In the code sample, lines 1 through 3 setup the timestamp that is passed into *delay_until*.

*Get_time* is used to obtain the current platform time, and an offset of 400 nanoseconds is added to the timestamp with *adds* and *adc* instructions. The *adds* instruction does a 32 bit add and saves the carry bit in the processor state register, so *adc* can use the carry along with its 32 bit addition. The 400 nanosecond offset added to the timestamp is the minimum execution time specified for the code between *get_time* and *delay_until*. This also includes time it takes to compute the deadline timestamp, as both *adds* and *adc* instructions execute between *get_time* and *delay_until*. When the *delay_until* instruction is decoded, the deadline timestamp is checked against platform time. The program will be delayed until platform time passes the deadline timestamp. If platform time has already passed the deadline timestamp, then this *delay_until* instruction will simply act as a *nop*, and the program will continue to execute.

It is important to know that *delay_until* merely specifies a minimum execution time. If the execution of the code block takes longer than the specified offset to execution, *delay_until* will have no effect on the program. Thus, *delay_until* should not be used to enforce real-time constraints. Instead, *delay_until* can be used to synchronize programs with external sources. For example, the VGA controller presented in [45] is implemented with the same mechanics to send the horizontal and vertical sync signals to the monitor from software. In chapter 4 we will also show applications that use this mechanism to synchronize the communication of hardware threads, and remove the execution time variance exhibited by software control paths.

### Exception_on_Expire and Deactivate_Exception

*Delay_until* and *delay_and_set* are only used specify minimum execution times, and cannot express a desired maximum execution time for code blocks. The *exception_on_expire* instruction is introduced to for this purpose, to specify a desired maximum execution time for code blocks. A new exception is added to the ARM exception vector table which represents a timer expired exception. *Exception_on_expire* takes as input a 64 bit timestamp. When *exception_on_expire* is decoded, the timestamp is registered as the timeout value. When platform time exceeds the timeout value, the timer expired exception is thrown in hardware, and the corresponding entry in the exception vector table is executed. The *deactivate_exception* instruction takes no input, and is simply used to deactivate the timeout value in hardware before an exception is thrown. When *deactivate_exception* is decoded, any timeout value that is currently registered by *exception_on_expire* is deactivated, and no timer expired exception will be thrown. Listing 3.3 shows the sample assembly code of using *exception_on_expire* with *deactivate_exception*.

Listing 2.2: Sample assembly code of exception_on_expire and deactivate_exception

```
1  cdp p13, 8, c2, c0, c1, 0  ; get_time
2  adds c3, c3, #400
3  adc c2, c2, #0
4  cdp p13, 2, c2, c2, c3, 0  ;exception_on_expire
5
6  add r5, r6, r6             ; code block that is executed
7  add r7, r5, r6
8
9  cdp p13, 5, c0, c0, c0, 0  ;deactivate_exception
10 b end
```

In the code sample, line 1 to 3 is used to setup the timestamp that is passed into *excep-*

*tion_on_expire*. It uses *get_time* and then adds an offset to the timestamp obtained. Line 4 passes the timestamp to *exception_on_expire*, which stores it to be checked in hardware. If the platform time were to exceed the the timestamp during execution of lines 6 and 7, which signifies a missed deadline, then a timer expired exception would trigger in hardware, and the control flow would jump to the exception handler. Or else, the *deactivate_exception* instruction on line 9 would deactivate that timestamp, and the program would continue to execute.

Currently only one timeout value is kept in hardware as part of the processor state. This means that at any moment in time, only one timestamp value can be stored and checked in hardware. Multiple deadlines can be managed in software, using data structures to keep an ordered list of deadlines to be checked. Multiple timeout slots can be implemented and checked in hardware at the cost of hardware complexity.

Similar to *delay_and_set* and *delay_until*, *exception_on_expire* and *deactivate_exception* merely create a mechanism to specify desired timing constraints. None of the timing instructions enforce execution time behavior, they merely provide a method for users to monitor, detect, and interact with the timing variability in software. This is in line with our original goal, to introduce timing semantics to the ISA without over-constraining the temporal properties of the ISA. These instructions do not limit the improvement of performance in the architecture for other instructions, as long as the timing properties of the timing instructions are faithfully implemented. With the introduction of these timing instructions, programmers can reason and control temporal properties of the program with timing instructions, independently of the architecture. At the same time, these instructions by themselves do not provide guarantees on the execution time of programs. An underlying architecture must still provide predictable execution times in order to for static analysis to guarantee a worst-case execution time.

### 2.3.2 Example Usage

In this section we show different use cases for the timing instructions introduced. We demonstrate different timing behaviors that can be built with the timing instructions to show how the assembly level instructions can be used by higher level languages to synthesize different timing behaviors.

#### Constructing Different Timing Behaviors

First we show various methods of constructing different timing behaviors from a code block. The code block can be a task, a function, or any piece of code that might exhibit timing variability. Here we simply refer to this code block as a task. We assume there is a desired execution time for this code block. The desired execution time could be from a specification of the application, or a synthesized timing requirement from a higher level model. We will call this desired execution time the deadline of the task. Figure 2.14 shows four possible timing behaviors that we can construct for this task using the assembly level instructions.

If the actual execution time of the task is longer than the specified deadline, the deadline is missed. Two possible timing behaviors can be used to handle this situation, which we show in scenario A and B in figure 2.14. Scenario A is used if the execution of task needs to completed. It could be that the task modifies external I/O states which cannot afford to be left in an unknown

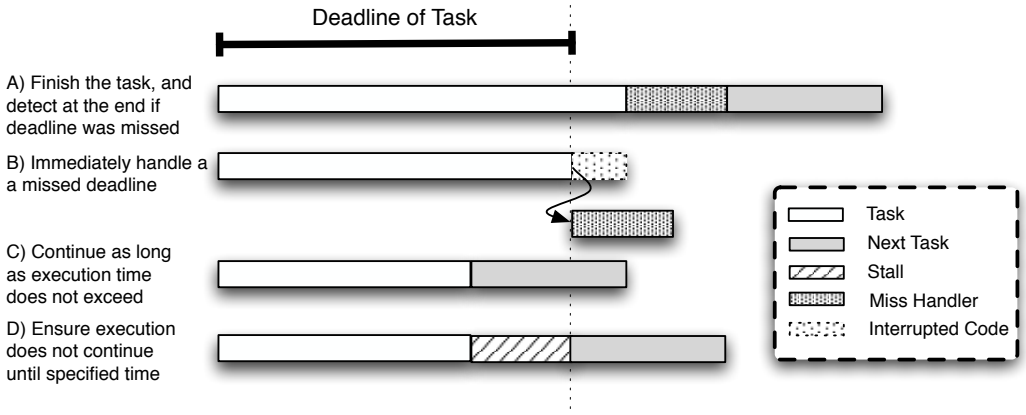



Figure 2.14: Different Desired Timing Behaviors

state. In this case, the task is first completed, then a miss handler is executed, and then the next task continues execution. This is also known as a *late miss detection*. Listing 2.3 shows how this is implemented using our timing instructions. Lines 1 to 3 of the listing is used to setup the deadline timestamp, which is stored in r2 and r3. Line 5 branches to the task and returns when the task completes. Lines 7 to 10 is where the miss detection occurs. We simply use another *get_time* instruction to obtain the current platform time and compare it with the deadline timestamp. The *blmi* instruction is a *branch with link* instruction that is conditionally executed only if the *[N]egative* condition code is set. Thus, the branch to *miss_handler* is only executed if the deadline timestamp is less than the current platform time, which means the deadline was missed.

Listing 2.3: Assembly code to implement scenario A

```
1  cdp p13, 8, c2, c0, c0, 0  ; get_time, current timestamp stored in [c2, c3]
2  adds r3, r3, #0xDEAD       ; assuming the deadline is #0xDEAD
3  adc r2, r2, #0             ; lines 2 and 3 calculate the deadline timestamp
4
5  bl task                    ; execute Task
6
7  cdp p13, 8, c4, c0, c0, 0  ; get_time, current timestamp stored in [c4, c5]
8  subs r3, r3, r5            ; lines 8 and 9 check for deadline miss
9  sbc  r2, r2, r4            ;
10 blmi miss_handler          ; branch to miss_handler if negative
11                            ; condition code is set
12
13 bl task2                   ; execute next task
```

If the missed deadline is to be handled immediate, then we cannot check the deadline timestamp in software, but it must be checked in hardware. The *exception_on_expire* and *deactivate_exception* instructions are then used to immediately execute the *miss_handler* when the timer expires. This is shown as scenario B in figure 2.14. Listing 2.4 shows the usage of *exception_on_expire* and *deactivate_exception* to achieve this timing behavior. The code is similar the one showed in listing 3.3 for the example usage of *exception_on_expire* and *deactivate_exception*. In this case, if the *deactivate_exception* is not executed before platform time exceeds the deadline timestamp, then the deadline is missed and the timer expired exception is thrown in hardware. In

the listing we assume that *miss_handler* has been registered as the exception handler, and will be executed when the timer expired exception is thrown. The *miss_handler* can directly abort the finishing of task 1 and directly start task 2, or it could return to the program point where the exception was thrown and continue execution after the *miss_handler*. This is application dependent, and both can be supported in software.

Listing 2.4: Assembly code to implement scenario B and C

```
1  cdp p13, 8, c2, c0, c0, 0  ; get_time, current timestamp stored in [c2, c3]
2  adds r3, r3, #0xDEAD       ; assuming the deadline is #0xDEAD
3  adc r2, r2, #0             ; lines 2 and 3 calculate the deadline timestamp
4  cdp p13, 2, c2, c2, c3, 0  ; exception_on_expire, register [c2, c3]
5
6  bl task                    ; execute Task
7
8  cdp p13, 5, c0, c0, c0, 0  ; deactivate_exception
9
10 bl task2                   ; execute next task
```

When the execution time of the task does not exceed the specified deadline, two different behaviors can also be implemented. The first is shown in scenario C of figure 2.14, where the next task immediately begins to execute. In this scenario, we merely want to ensure that the task does not exceed the deadline. The code shown in the previous listing 2.4 exhibits this behavior. Once the task finishes earlier, *deactivate_exception* is executed to deactivate the exception, and the next task is immediately executed.

However, if we do not want the next task to start until after the specified deadline, then a *delay_until* can be used to ensure a minimum execution time for the task. This could be useful if the tasks are synchronized to an external source. The sample code is shown in listing 2.5, which is scenario D in figure 2.14.

Listing 2.5: Assembly code to implement scenario D

```
1  cdp p13, 8, c2, c0, c0, 0  ; get_time, current timestamp stored in [c2, c3]
2  adds r3, r3, #0xDEAD       ; assuming the deadline is #0xDEAD
3  adc r2, r2, #0             ; lines 2 and 3 calculate the deadline timestamp
4  cdp p13, 2, c2, c2, c3, 0  ; exception_on_expire, register [c2, c3]
5
6  bl task                    ; execute Task
7
8  cdp p13, 5, c0, c0, c0, 0  ; deactivate_exception
9  cdp p13, 4, c2, c2, c3, 0  ; delay_until
10
11 bl task2                   ; execute next task
```

The *delay_until* instruction is added after *deactivate_exception*, and whenever the execution time of the task is less than the specified deadline, it will delay the program until the deadline is reached, ensuring the next task will not execute early. The order of *delay_until* and *deactivate_exception* in this case is very important. If the order were the other way around, then *delay_until* would first delay the program until after the specified deadline. Because *deactivate_exception* has not executed yet, the timer expired exception would always be thrown, even if the task did not miss the deadline. Thus, *deactivate_exception* must be before *delay_until*. *Delay_until* can also be used

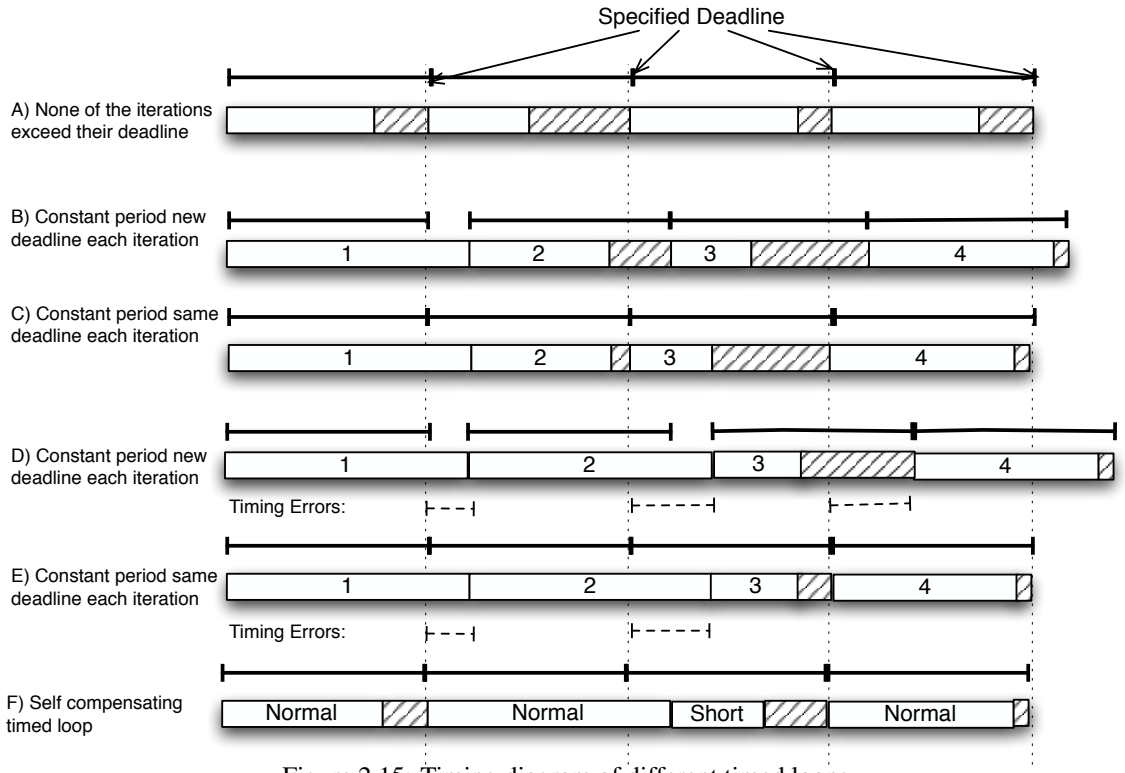



Figure 2.15: Timing diagram of different timed loops

in scenario A to achieve the same effect for late miss-detections. In that situation, simply insert a *delay_until* in line 12 of listing 2.3 and use the first deadline timestamp as its input.

### Timed Loops

By using timing instructions within loops, we can construct timed loops for programs that exhibit periodic timing behaviors. Listing 2.6 shows sample code that uses *get_time* and *delay_until* to construct a timed loop.

Listing 2.6: Timed loops with get_time and delay_until

```
1 loop:
2   cdp p13, 8, c2, c0, c0, 0   ; get_time, current timestamp stored in [c2, c3]
3   adds r3, r3, #0xDEAD        ; assuming the deadline is #0xDEAD
4   adc r2, r2, #0              ; lines 2 and 3 calculate the deadline timestamp
5
6   bl task                     ; execute Task
7
8   cdp p13, 4, c2, c2, c3, 0   ; delay_until
9   b loop
```

The period of each loop iteration is specified by the calculations of line 2 and 3 in listing 2.6, with the small additional time to execute the *get_time* and *delay_until* instruction. Ideally, the execution time of the task never exceeds the period of the loop, and the timing behavior shown in scenario A from figure 2.15 is observed. In this scenario, each iteration exhibits slightly differ-

ent execution times, but the *delay_until* instruction ensures each iteration takes the whole period to execute. However, if one iteration misses the deadline its execution time exceeds the period, then scenario B in figure 2.15 would be observed based on our current implementation. We see that iteration 1 is the only iteration that misses its deadline, but because *get_time* is called in the beginning of each loop iteration, our next deadline for iteration 2 will be shifted due to the overrun in execution time. Even though iteration 2 executes in less time, all future iterations are still shifted after one missed deadline.

The timestamps are stored in general purpose registers and can be manipulated using data-processing instructions, so we can modify slightly the implementation of the timed loop to account for that missed deadline. Listing 2.7 shows a different implementation of timed loops. In this implementation, we only call *get_time* once outside of the loop, and within the loop the deadline timestamps are incremented directly by arithmetic operations, shown on lines 3 and 4.

Listing 2.7: Timed loops with get_time outside of the loop

```
1   cdp p13, 8, c2, c0, c0, 0   ; get_time, current timestamp stored in [c2, c3]
2 loop:
3   adds r3, r3, #0xDEAD        ; assuming the deadline is #0xDEAD
4   adc r2, r2, #0              ; lines 3 and 4 calculate the deadline timestamp
5
6   bl task                     ; execute Task
7
8   cdp p13, 4, c2, c2, c3, 0   ; delay_until
9   b loop
```

Figure 2.15 scenario C shows the effects of this implementation. Although iteration 1 misses its deadline, but the execution time of iteration 2 is short enough to "make up" the delayed time cause from the first iteration. Future iterations are not effected by the missed deadline from iteration 1, and continue to execute as desired. By placing *get_time* outside of the loop, the increments to the deadline timestamp is purely the period of the loop, since we do not call *get_time* again obtain the current time. Of course, both implementations are susceptible to the effects of multiple missed deadlines in a row, as shown in scenario D and E. In this case, both iterations 1 and 2 overrun their deadline, and the timing error is compounded. With our first implementation of timed loops, the error jitter continues to increase, because the new deadline is set according to the late execution of each iteration, as shown in scenario D. The error jitter never recovers, even though iteration 3's execution time is short enough to allow recovery. As shown in scenario E, our second iteration recovers the period on the 3rd iteration, and the 4th iteration is not effected.

Furthermore, we can construct a timed loop that self compensates whenever it detects that an iteration overran its deadline. We do so by using the late miss detection mechanism shown previously in our timed loop to run a shorter version of the task whenever a previous deadline is missed. This is shown in listing 2.8.

In this sample code, we place the late miss detection in the beginning of each loop, and use it to detect if the current platform time is greater than the previously set deadline timestamp. On lines 4 and 5 we subtract an offset that is used to compensate for the execution time of the loop overhead and miss detection. This is an important step that cannot be omitted. For each iteration, if the previous iteration meets its deadline, the *delay_until* instruction will delay program execution until the current platform time exceeds the specified deadline. Thus, if the time it takes to execute

Listing 2.8: Timed loops with compensation

```
1   cdp p13, 8, c2, c0, c0, 0  ; get_time, deadline timestamp stored in [c2, c3]
2 loop:
3   cdp p13, 8, c4, c0, c0, 0  ; get_time, current timestamp stored in [c4, c5]
4   subs r5, r5, #<offset>     ; <offset> is implementation dependent and used to
5   sbc  r4, r4, #0            ; account for loop overhead and miss detection
6
7   subs r3, r3, r5            ; Check if previous iteration deadline is missed
8   sbc  r2, r2, r4            ;
9
10  blmi task_short            ; execute shorter task if previous deadline mess
11  blpl task_normal           ; or else execute normal task
12
13  adds r3, r3, #0xDEAD       ; assuming the deadline is #0xDEAD
14  adc r2, r2, #0             ; calculate the deadline timestamp for this iter.
15  cdp p13, 4, c2, c2, c3, 0  ; delay_until
16
17  b loop
```

the loop overhead and miss-detection is not accounted for, then we will always detect a missed deadline from the effects of *delay_until*. The actual offset is implementation dependent, depending on how long each instruction takes to execute. We will show how this offset is calculated in our implementation in section 3.6.3. Once the overhead is accounted for, lines 7 and 8 check if the previous deadline was met, and lines 10 and 11 executes the short task if the deadline was missed, or executes the normal task else wise. In this case, we delay the deadline calculation for this iteration until right before the *delay_until* instruction, because the miss detection checks against the previous deadline timestamp. The timing behavior that is created is shown in figure 2.15 scenario F.

Other combinations of timing instructions can further be explored. For example, the use of *exception_on_expire* and *deactivate_exception* to handle cases where loop iterations exceeds the period. In these examples, we are not claiming that a particular implementation of timed loops is the "correct" implementation. We mainly show different possible ways to implement a timed loop construct with our timing extensions, and point out some subtleties when doing so.

# Chapter 3

# Precision Timed ARM

The Precision Timed ARM (PTARM) architecture is a realization of the PRET principles on the ARM instruction set architecture [12]. In this chapter we describe in detail the implementation of the timing-predictable ARM processor and the timing analysis on the architecture. We show that with the architectural design principles of PRET, the PTARM architecture is easy analyzable with repeatable timing.

The architecture of PTARM follows the design principles discussed in chapter 2. This includes a thread-interleaved pipeline and an exposed memory hierarchy with scratchpads and a timing predictable DRAM controller. The ARM ISA was chosen not only due to its popularity in the embedded community, but also because it is a *Reduced Instruction Set Computer* (RISC), which contains simpler instructions that allow more precise timing analysis. *Complex Instruction Set Computers* (CSIC), such as Intel's x86 ISA, adds complexity to the instructions, hardware, and timing analysis. RISC architectures typically feature a large uniform register file, use a load/store architecture, and use fixed-length instructions. In addition, the ARM ISA contains several unique features. Here we list of a few. First, the ARM ISA does not contain explicit shifting instructions. Instead, data-processing instructions can shift its operands before the data operation. This requires a separate hardware shifter in addition to the arithmetic logic unit (ALU) in the hardware. Second, ARM's load/store instructions contain auto-increment capabilities that can increment or decrement the value stored in the base address register. This is done when load/store instructions use the pre or post-index addressing mode. This is useful to compact code that operate on data structures such as arrays or stacks. In addition, almost all of the ARM instructions are conditionally executed. The conditional execution improves architecture throughput with potential added benefits of code compaction [27].

ARM programmer's model specifies 16 general purpose registers (R0 to R15), with register 15 being the program counter (PC). Writing to R15 triggers a branch to the written value, and reading from R15 reads the current PC plus 8. ARM has a rich history of versions for their ISA. PTARM implements the ARMv4 ISA, without support for the thumb mode. In addition to the predictable architecture, PTARM extends the ARM ISA with timing instructions introduced in chapter 2.3. We describe the implementation of these timing instructions in detail in section 3.4.4 below.

## 3.1 Thread-Interleaved Pipeline

PTARM implements a thread-interleaved pipeline for the ARM instruction set. Currently, PTARM targets Xilinx Virtex-5 Family FPGAs, thus several design decisions were made to optimize PTARM for that FPGA family. PTARM implements a 32 bit datapath five stage thread-interleaved pipeline. Thread-interleaved pipelines remove pipeline hazards with by interleaving multiple threads, improving throughput and predictability. Conventional thread-interleaved pipelines have at least as many threads as pipeline stages to keep the pipeline design simple and maximize the clock speed. However, Lee and Messerschmitt [58] showed that hazards can also be removed in the pipeline even if the number of threads is one less than the number of pipeline stages. Increasing the number of threads in the pipeline increases each thread's latency, because threads are time-sharing the pipeline resource. Thus, PTARM implements a five stage thread-interleaved pipeline with four threads by carefully designing the PC writeback mechanism one pipeline stage earlier.

Figure 3.1 shows a block diagram view of the pipeline. Some multiplexers within the pipeline have been omitted for a clearer view of the hardware components that make up the pipeline. There contains four copies of the Program Counter(PC), Thread States, and Register File. The register file has 3 read ports and 1 write port. Most of the pipeline design follow the five stage pipeline described in Hennessey and Patterson [41], with the five stages in the pipeline being *Fetch*, *Decode*, *Execute*, *Memory*, and *Writeback*. We briefly describe the functionality of each stage, and leave more details to section 3.4, where the instruction implementations are presented.

The *fetch stage* of the pipeline selects the correct PC according to which thread is executing, and passes the address to instruction memory. A simple 2 bit ($log(n)$) up-counter is used to keep track of which thread current to fetch. This reduces the time and space overhead of context switching close to zero. The PC forward path is used when an instruction loads to R15, which causes a branch to the value loaded from main memory. We will discuss the need for the forwarding path below when the *memory stage* is described. The *timer* implements the *platform clock* used by the timing instructions. In addition, it contains the hardware logic that registers and checks for timer expiration exceptions for each thread. A 64 bit timestamp in nanoseconds is associated with each instruction when it begins execution in the pipeline. This 64 bit timestamp is latched together from the *timer* in the fetch stage, and is kept with the instruction for the duration of its execution.

The *decode stage* contains the *pipeline controller* that decodes instructions and deter-

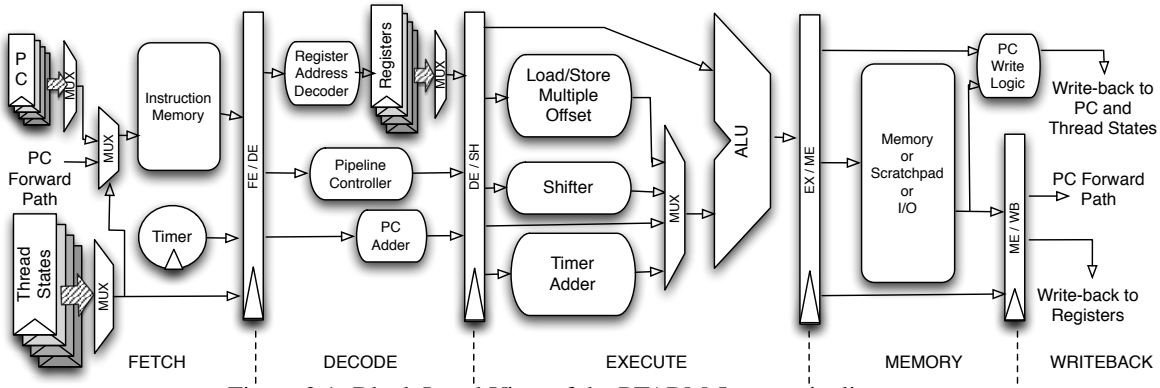



Figure 3.1: Block Level View of the PTARM 5 stage pipeline

mines the pipeline control signals to be propagated down the pipeline. Most of ARM instructions are conditionally executed, so the pipeline controller also checks the condition bits against the processor state condition codes to determine whether the instruction is to be executed or not. Typically, *pipeline controllers* needs to keep track of all instructions currently executing in the pipeline, to detect the possibility of pipeline hazards and handle them correspondingly. However, from the *decode* stage of our thread-interleaved pipeline, other instructions executing in the pipeline are instructions from other threads. Thus, the controller logic is greatly simplified because no hazard checking from in-flight instructions is required. A small decoding logic, the *register address decoder*, is inserted in parallel with the controller to decode the register addresses from the instruction bits. In typical RISC instruction sets, such as MIPS, the register operands have a fixed location for all instruction encodings. Thus, they can directly be passed into the register file before decoding. However, in the ARM instruction set, certain instructions encode the register read address at different bit locations of the instruction. For example, data-processing register shift instructions and store instructions reads a third operand from the register that are at encoded at different bit locations. Thus, a small register address decoding logic is inserted for a quick decoding of the register addresses from the instruction bits.

The *PC Adder* is the logic block that increments the PC. Single threaded pipelines need to increment the PC immediately in the fetch stage to prepare for instruction fetch the next processor cycle. For thread-interleaved pipelines, the next PC from the current thread is not needed until several cycles later, so there is no such restriction. In addition to outputting the current PC incremented by 4, the *PC Adder* also outputs the value of the current PC incremented by 8. In the ARM ISA, instructions that use R15 as an operand actually read the instruction PC plus 8, instead of the instruction PC, as the value of the operand. This is designed for the convenience of architecture implementation. Typically in pipelines, instructions take 2 cycles (fetch and decode) before it enters the execute stage. For single-threaded pipelines, the program counter has likely been incremented by 8 at the time the instruction enters the execute stage. By using $instruction\_pc + 8$ as the operand value, the hardware implementation can directly use the processor PC currently in the fetch stage, without compensating for the two increments that occurred. However, for thread-interleaved pipelines, we need to explicitly calculate $instruction\_pc + 8$, because the PC for each thread is not incremented every processor cycle, but incremented once every round-robin scheduling cycle. Since $instruction\_pc+8$ could be used as a data operand needed in the execute stage, the *PC Adder* in placed in the *decode* stage.

The *execute stage* contains execution units and multiplexers that select the correct operand and feeds it to the ALU. The ARM ISA assumes an additional shifter to shift the operands before data operations, so a 32 bit *Shifter* is included. The 32 bit *ALU* does most of the logical and arithmetic operations, including data-processing operations and branch address calculations. The *Load/-Store Multiple Offset* logic block calculates the offset for load/store multiple instructions. Load/store multiple instructions use a 16 bit vector to represent each of the 16 general purpose registers. Memory operations are done only on the registers whose corresponding bit value is set in the bit vector. The memory addresses of each memory operation is derived from the base register and an offset. The *Load/Store Multiple Offset* logic block calculates this offset according to the bit count of the remaining bit vector during load/store multiple instructions. The *Timer Adder* is a 32 bit add/subtract unit used with the *ALU* to compare 64 bit timestamps for timing instructions. Specifically, *delay_until* requires the comparison of two 64 bit timestamps every thread cycle, thus the additional

*Timer Adder* is added to accomplish that. The implementation details of *delay_until* is described in section 3.4.4.

The *memory stage* issues the memory operations and writes back the PC and thread states. The PC and thread states are written back a stage early to allow us to interleave four threads in our five stage pipeline, instead of five. This improves the latency of individual threads. When four threads are interleaved through a five stage pipeline, if the PC is written back in the *writeback stage*, then the next instruction fetch for the thread would not see the updated PC in time for its instruction fetch. Figure 3.2 illustrates this by showing an execution sequence of the four thread five stage thread-interleaved pipeline in PTARM. Each cycle, the instructions in the *fetch* and *writeback* stages belong to the same thread. Thus, committing the PC in the *writeback stage* would cause a control hazard because the updated PC would not observed by the concurrent instruction fetch. For most instructions, including branch instructions, the next PC is known before the memory stage, so moving the PC commit one stage earlier does not cause any problems. The *PC Write Logic* updates the next PC, depending on the instruction, and whether an exception occurred or not. Section 3.3 describes the hardware mechanism for handling exceptions in PTARM. Normally, PC+4 from the *PC Adder* or the results from the *ALU* is used to update the PC.

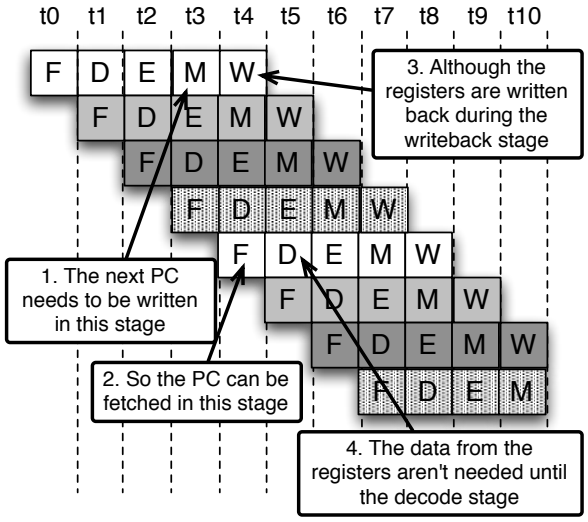



Figure 3.2: Four thread execution in PTARM

Whenever instructions write to R15 (PC), the control flow of the program branches to the value written to R15. Data processing instructions that write to R15 have its results computed by the *execute stage*, ready to be committed as the new PC in the *memory stage*. However, a load instruction that loads to R15 will not know the branch target until after the memory read. Thus, a PC forwarding path is added to forward the results back from memory as the fetched PC if a load instruction loads to R15. The forwarding path does not cause any timing analysis difficulties because the statically the forwarding path is only and always used when a load instruction loads to R15, which can be statically determined. Also, this causes no stall in the pipeline, and does not effect the timing of any following instructions. We describe the implementation more details in section 3.4.3.

The *writeback stage* simply writes back the results from memory or the *ALU* to the correct registers. Writing back to registers in the *writeback stage* does not cause data hazards even if there are only four threads, because the data from registers are not read until the following *decode stage*. Figure 3.2 shows that the two stages do not overlap in the same cycle, thus causing no hazards.

## 3.2 Memory Hierarchy

The memory hierarchy of PTARM is exposed in software, as discussed in section 2.2. This allows for a more predictable and analyzable memory access latency from the architecture.

The memory hierarchy is composed of regions reserved for the boot code, the instruction and data scratchpads, a 512MB DRAM Module, and the memory mapped I/O, all occupying separate address regions. Figure 3.3 shows the memory address regions reserved for each memory type. Both the boot code and scratchpads are synthesized to dual-ported block RAMs on the FPGA, and provide deterministic single cycle access latencies.

### 3.2.1 Boot code

The *boot code* region contains initialization and setup code for PTARM. This includes the exception vector table, which stores entries used jump to specific exception handlers for the different exceptions. The specific table entries and layout is explained in section 3.3. Non-user registered exception handlers and the exception setup code are also part of the boot code . When PTARM resets, all threads begin execution at address 0x0, which is the *reset* exception entry in the exception vector table. The *reset* exception handler will setup each thread's execution state, including the stack, which is allocated on the data scratchpad. Then the handler transfers control flow to the user compiled code for each thread. Dedicated locations in the boot code is reserved for user-registered exception handlers, these entries can be modified programmatically. For example, a location is reserved to store the location of a user registered timer expire exception handler.

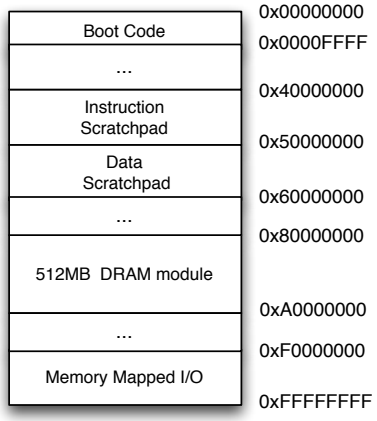



Figure 3.3: Memory Layout of PTARM

### 3.2.2 Scratchpads

The scratchpads replace caches as the fast-access memory in our memory hierarchy. The partition of instruction and data scratchpads between threads can be configured into different schemes depending on the application. For embedded security applications, such as encryption algorithms, partitioning the scratchpads into private regions in hardware for each thread might be desired to prevent cross-thread attacks. In section 4.2 we discuss the security implications and how partitioning the scratchpad can defend against timing side-channel attacks that exploit underlying shared resources. On the other hand, on applications with collaborative hardware threads, sharing the scratchpad could provide flexibility for the memory allocation scheme [102] of scratchpads and communication between hardware threads. This opens opportunities to optimize system performance, instead of just individual threads. Hybrid schemes can also be used that privatizes a hardware thread for security, and allows other threads for collaboration.

### 3.2.3 DRAM

PTARM interfaces with a $512MB$ DDR2 $667MHZ$ DRAM memory module (Hynix HYMP564S64CP6-Y5). All access to the DRAM goes through the predictable DRAM controller described in section 2.2.2. The DRAM controller privatizes the DRAM banks into four resources, which we assign to each thread in our pipeline. This removes bank access conflicts and gives us

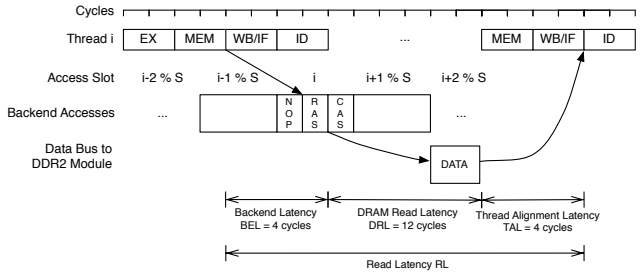



Figure 3.4: Example load by thread $i$ in the thread-interleaved pipeline.

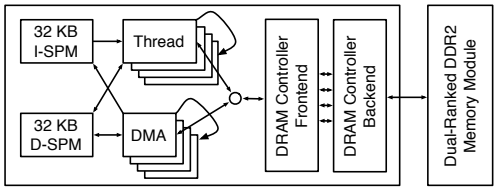



Figure 3.5: Integration of PTARM core with DMA units, PRET memory controller and dual-ranked DIMM [89].

predictable memory access latencies to the DRAM. The pipeline interacts with the frontend of the DRAM controller, which routes requests to the correct request buffer in the backend, and manages the insertion of row-access refreshes to ensure the refresh constraint is met. In conventional memory architectures where the hierarchy is hidden, the processor interacts with DRAM indirectly by the filling and writing back of cache lines. In our memory system, the processor can directly access the DRAM through load and store instructions to the distinct memory regions of the DRAM. In addition, each hardware thread is also equipped with a direct memory access (DMA) unit, which can perform bulk transfers between the scratchpads and the DRAM. Figure 3.5 shows the integration of PTARM with the DMA units, memory controller and DRAM.

When DRAM is accessed through load (read) and store (write) instructions, the memory requests are issued directly from the memory stage of pipeline. The request is received from the frontend of the memory controller, and placed in the correct request buffer. Depending on the alignment of the pipeline and the backend, it takes a varying number of cycles until the backend generates corresponding commands to be sent to the DRAM module. After the read has been performed by the DRAM and has been put into the response buffer, again, depending on the alignment of the pipeline and the backend, it takes a varying number of cycles for the corresponding hardware thread to pick up the response. Figure 3.4, illustrates the stages of the execution of an example read instruction in the pipeline. In [89] we derive the access latencies from the alignment and show that it is either 3 or 4 thread cycles. We leverage this misalignment of the pipeline and backend to hide the refresh latency from the front end. When a refresh is scheduled for the DRAM resource, if no memory request is in the request buffer, the refresh is serviced. As mentioned in section 2.2.2, if a refresh conflicts with a pipeline load or store, we push back the refresh till after the load or store. In this case, the pushed back refreshes become invisible: as the pipeline is waiting for the data to be returned and takes some time to reach the memory stage of the next instruction, it is not able to use successive access slots of the backend, and thus it does not observe the refresh.

Whenever a DMA transfer is initiated, the DMA unit uses the thread's request buffer slot to service DMA requests to/from the scratchpad. Thus, while a DMA transfer is initiated, the thread gives up access to the DRAM to the DMA unit. During this time, the thread can continue to execute and access the scratchpad regions that is not being serviced by the DMA request. This is possible because scratchpads are dual-ported, allowing a DMA unit to access the scratchpads simultaneously as its corresponding hardware thread. If at any point the thread tries to access the DRAM, it will be blocked until the DMA transfer completes. Similarly, accesses to the region of the scratchpad

being serviced by the DMA will also stall the hardware thread[1]. The DMA units can fully utilize the bandwidth provided by the backend because unlike accesses from the pipeline, they suffer the no alignment losses. When refreshes conflict with a DMA transfer, we push back the first refresh and schedule one at the end of the DMA transfer. This can be seen as shifting all refreshes, during the DMA transfer, back by 63 slots or to the end of the transfer. More sophisticated schemes would be possible, however, we believe their benefit would be slim. With this scheme, refreshes scheduled within DMA transfers are predictable so the latency effects of the refresh can be easily analyzed, which we derive in [89].

**Store Buffer** Stores are fundamentally different from loads in that a hardware thread does not have to wait until the store has been performed in memory. By adding a single-place store buffer to the frontend, we can usually hide the store latency from the pipeline. Using the store buffer, stores to DRAM that are not preceded by other memory operations to DRAM can appear to execute in a single thread cycle. Otherwise, the store will observe the full two thread cycle latency to store to DRAM. A bigger store buffer would be able to hide latencies of more successive stores at the expense of slightly increasing the complexity of timing analysis.

### 3.2.4 Memory Mapped I/O

Currently PTARM implements a primitive I/O bus for communicating with external inputs and outputs. Access to the bus occurs in the memory stage of the pipeline, by accessing the memory mapped I/O region with memory instructions. I/O devices snoop the address bus to determine whether the pipeline is communicating with it. The I/O bus is shared by all threads in the thread-interleaved pipeline, thus, in addition to address and data, a thread ID is also sent out for potential thread-aware I/O devices. In section 3.5.1 below we describe the several I/O components that are connected to our PTARM core. Currently all I/O devices interface with the processor through single cycle memory mapped I/O control registers to prevent bus contention between threads. In order to ensure predictable access times to all I/O devices, a timing predictable bus architecture must be used [119]. A predictable thread-aware I/O controller is also needed to ensure data from the I/O devices are read by the correct thread, and contention is properly managed. These issues present future research opportunities – to interface a timing predictable architecture with various I/O devices while maintaining its timing predictability.

## 3.3 Exceptions

When exceptions occur in a single threaded pipeline, the whole pipeline must be flushed because of the control flow shift in the program. The existing instructions in the pipeline immediately become invalid, and the pipeline fetches instructions from an entry in the exception vector table. The exception vector table stores entries that direct the control flow to the correct exception handling code. The table is part of the boot code, and its contents are shown in table 3.1. The timer expired exception entry is added with our timing extensions to the ISA, and is triggered when a user registered timestamp with *exception_on_expire* expires.

[1] This does not affect the execution of any of the other hardware threads.

| Address | Exception Type | Description |
|---|---|---|
| 0x0 | Reset | Occurs when the processor resets |
| 0x4 | Undefined instructions | Occurs when an undefined instruction is decoded |
| 0x8 | Software Interrupt (SWI) | Occurs when a SWI instruction is decoded |
| 0x18 | Interrupt (IRQ) | Occurs on external interrupts |
| 0x1C | Timer Expired | Occurs when a thread's exception timer expires |

Table 3.1: Exception vector table in PTARM

In the PTARM thread-interleaved pipeline, exceptions are separately managed for each hardware thread. All threads are designed to be temporally isolated in the PTARM thread-interleaved pipeline. Thus, an exception that triggers on one thread must not effect the execution of other threads in the pipeline. In PTARM, any exceptions that occur during execution propagate down the pipeline with the instruction. The exception is checked and handled before modifying any states, such as the PC, CPSR, register, memory, of the thread. When an exception is detected, the current instruction execution is ignored, and the PC and thread states are updated to handle the exception. According to the exception type, the PC is redirected to the corresponding entry in the exception vector table. The current PC is to store in the link register (R14), so the program can re-execute the halted instruction if desired.

None of the other instructions executing in the pipeline are flushed when an exception occurs. As shown in figure 3.6, the instructions executing in other pipeline stages belong to other threads, so no flushing of the pipeline is required because no instruction was speculatively executed. This limits the timing affects of exceptions to only one thread, as the timing behavior of other threads in the pipeline are unaffected. From the hardware, only a one thread cycle overhead is induced. In this thread cycle, the current instruction does not complete its execution, but instead the pipeline updates the thread states to reflect the exception. In the next thread cycle, the thread will already be executing instructions to handle the exception.

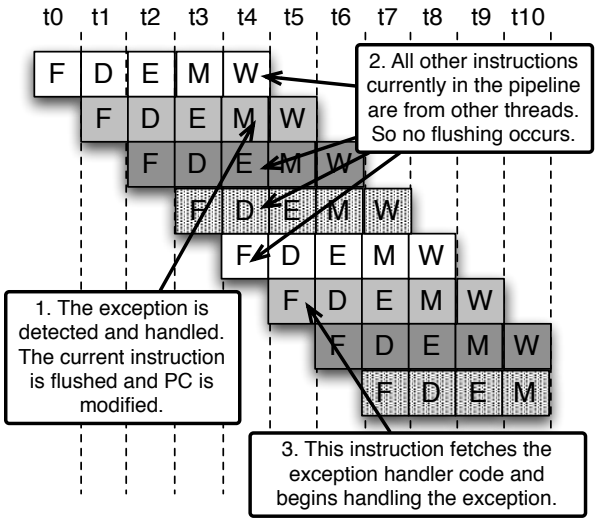



Figure 3.6: Handling Exceptions in PTARM

For longer latency instructions that modify the program state, exceptions can cause an inconsistent view of the program state when an exception occurs during execution. For example, a *timer_expired* exception could occur in the middle of a memory instruction to the DRAM region. In this case, we cannot cancel the memory request abruptly because the memory request is handled by the external DRAM controller, and possibly already being serviced by the DRAM. If the memory instruction is a load, the results can be simply disregard. But if the instruction is a store instruction, we cannot cancel the store request that is writing data to the memory. The programmer must disable interrupts before writing to critical

memory locations that require a consistent program state.

Besides an inconsistent program state, interrupting a memory instruction can also complicate the interaction between the pipeline and DRAM controller. The DRAM controller, with a request buffer of size one, does not queue up memory requests. This normally is not an issue because our pipeline does not reorder instructions or speculatively execute when there are outstanding memory requests. However, if a memory instruction is interrupted, the pipeline flushes the current instruction, and control flow directly jumps the exception vector table, which directs the program to execute the corresponding exception handler. If instructions immediately following the exception access the DRAM, a new memory request would be issued to the DRAM controller that is still servicing the previous request prior to the exception. The new memory request would then need to be queued until the previous "canceled" memory request completes before it can begin being serviced. This creates timing variability for exception handlers, because the latency of initial load instructions would vary depending on the instruction interrupted by the exception. Because it is very difficult to statically analyze the exact instruction an exception would interrupt, it will be difficult to predict when this timing variance would occur.

To achieve predictable and repeatable timing for exception handlers, we leverage the exposed memory hierarchy to ensure sufficient time has lapsed for the DRAM controller to finish servicing any potential memory requests, before any instructions in the exception handlers access the DRAM. In PTARM, the worst-case memory latency for the DRAM backend is 4 thread cycles, so we need to ensure that the instructions executed during the first 3 thread cycles after an exception does not access the DRAM. The exception vector table and the exception handler setup code are all part of the boot code synthesized to dual-ported BRAMs, thus instruction fetching is guaranteed to avoid the DRAM. The exception vector entries contain only contain branch instructions, which also does not access the DRAM. We statically compile the data stack onto the data scratchpad, so any stack manipulations that occurs also avoids the DRAM. Thus, the exception handling mechanism in PTARM is timing predictable and repeatable. In section 3.6.4 we will show an example to demonstrate this.

Currently PTARM does not implement an external interrupt controller to handle external interrupts. But when implementing such an interrupt controller, each thread should be able to register specific external interrupts that it handles. For example, a hard real-time task could be executing on one thread, while another task without timing constraints is executing on another thread waiting for an interrupt to signal the completion of a UART transfer. In this case, the thread running the hard real-time task should not be interrupted when the UART interrupt occurs. Only the specific thread handling the UART transfers should be interrupted by this interrupt. Thus, we envision a thread-aware interrupt controller that allows each thread to register specific interrupts to handle.

## 3.4 Instruction Details

In this section we present details on how each instruction type is implemented to show how each hardware block in the pipeline shown in figure 3.1 is used. We will go through different instruction types and discuss the timing implications of each instruction in our implementation.

### 3.4.1 Data-Processing

We begin by explaining how data-processing instructions are implemented. These instructions are used to manipulate register values by executing register to register operations. Most data-processing instructions take two operands. The first operand is always a register value. The second operand is the shifter operand, which could be an immediate or a register value. Both can be shifted to form the final operand that is fed into the ALU. Figure 3.7 explains how data-processing instructions are executed through the pipeline.

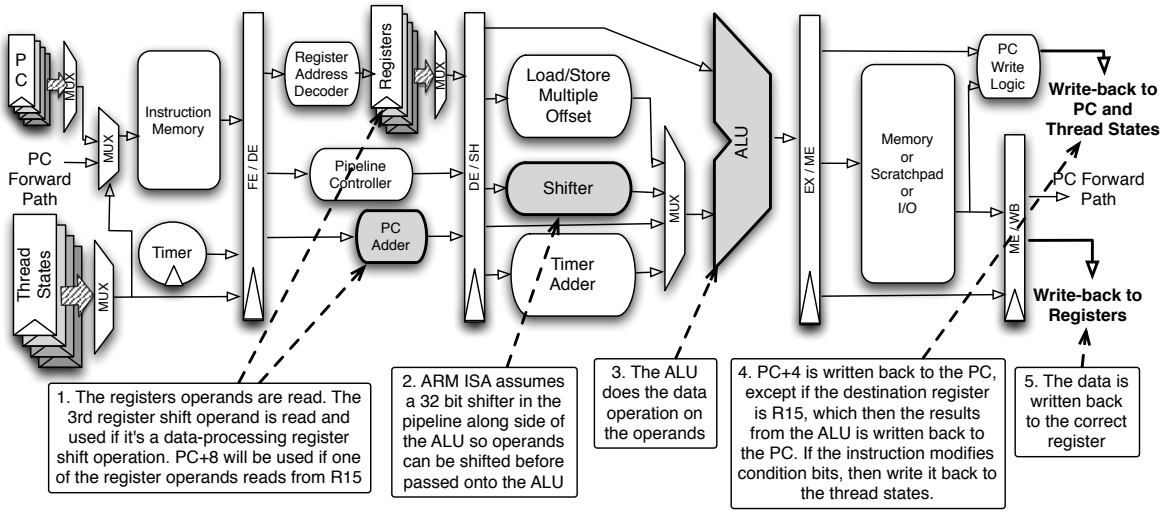



Figure 3.7: Data Processing Instruction Execution in the PTARM Pipeline

The execution of data-processing instructions is fairly straightforward. Operands are read from the register file or instructions bits. They are shifted if required, then sent to the ALU for the data operation. Because R15 is the PC, so instructions that use R15 as an operand will read the value of PC+8 as the operand. Any instruction that uses R15 as the destination register will trigger a branch, which simply writes back the results from the ALU to the next PC. Otherwise the are written back in the writeback stage.

Data processing instructions can also update the program condition code flags that are stored in the thread state. Some instructions that update the condition code flags do not writeback data to the registers, but only update the condition code flags. The condition code flags are used to predicate execution for ARM instructions. It consists of four bits: Zero (Z), Carry (C), Negative (N) and Overflow (V). The high four bits of each instruction forms a conditional field that is checked against the condition code flags in the pipeline controller to determine whether or not the instruction is executed.

All data-processing instructions only take one pass through the pipeline, even instructions that read from or write to R15. So all data-processing instructions take only one thread cycle to execute.

### 3.4.2 Branch

Branch instructions in the ARM can conditionally branch forward or backwards by up to 32MB. There is no explicit conditional branch instruction in ARM. Conditional branches are implemented using the ARM predicated instruction mechanism. Thus, the condition code flags determine if a conditional branch is taken or not Figure 3.8 shows how branch instructions are executed our the thread-interleaved pipeline.

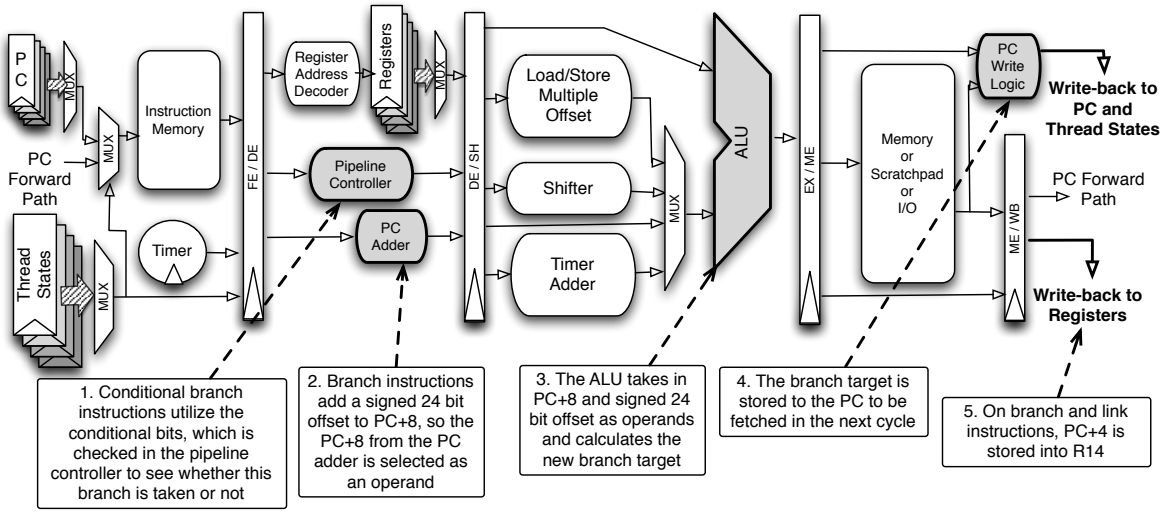



Figure 3.8: Branch Instruction Execution in the PTARM Pipeline

The branch instructions for the ARM ISA calculate the branch target address by adding a 24 bit signed offset, specified in the instruction, to the current PC incremented by 8. Thus, the PC+8 output from the PC Adder is used as an operand for the ALU to calulate the target branch address. Once the address is calculated, it is written back to its thread's next PC ready to be fetched. If the instruction is a branch and link ($bl$) instruction, PC+4 is propagated down the pipeline and written back to the link register (R14).

All branch instructions, whether conditionally taken or not, all take only one thread cycle to execute. But more importantly, the next instruction in the thread executed after the branch , whether it is a conditional branch or not, is not stalled or speculatively executed. Rather, it is fetched after the conditional branch is resolved, and the branch target address is calculated. The thread-interleaved pipeline simplifies the implementation of the branches and removes the need for control hazard handling logic. Instead of speculating the branch target address the next processor cycle, instructions from other threads will be fetched and executed.

### 3.4.3 Memory Instructions

There are two type of memory instructions implemented in PTARM from the ARM ISA: Load/Store Register and Load/Store Multiple. We discuss both type of memory instructions, and also present the special case when a load instruction loads to R15. This triggers a branch which loads the branch target address from memory. Although this slightly complicates our pipeline de-

sign, we show that it does not affect the timing predictability and execution of the instruction, and subsequent instructions after the triggered branch. Currently load/store halfword and doubleword are not implemented in PTARM, as they fall under the miscellaneous instructions category. These instructions can easily be implemented using the same principles described below.

### Load/Store Register

Load instructions load data from memory to registers, and store instructions store data from registers to memory. Store instructions utilizes the third register read port to read in the register value to be stored to memory. The memory address is formed by combining a base register and an offset value. The offset value can be a 12 bit immediate encoded from the instruction, or a register operand that can be shifted. The current load/store instructions supports word or byte operations. Figure 3.9 describes how load/store register is implemented in the pipeline.

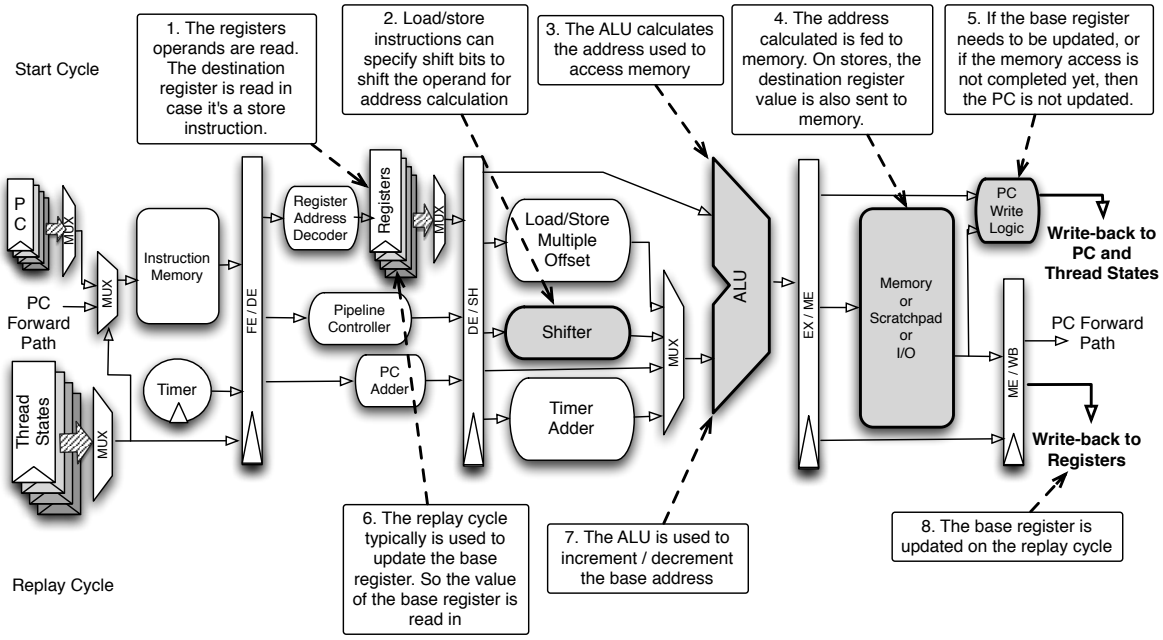



Figure 3.9: Load/Store Instruction Execution in the Ptarm Pipeline

Accesses to different memory regions yield different latencies for memory instructions. When the memory address accesses the scratchpad or boot code memory region, memory operations are completed in a single processor cycle. Thus, the data is ready in the following (*writeback*) stage to be written back to the registers. However, if the DRAM is accessed, the request must go through the DRAM memory controller, which takes either three or four thread cycles to complete. Our thread-interleaved pipeline implementation does not dynamically switch threads in and out of execution when they are stalled waiting for memory access to complete. Thus, when a memory instruction is waiting for the DRAM, the same instruction is replayed by withholding the update for the next PC, until the data from DRAM arrives and is ready to be written back in the next stage. The memory access latencies to the I/O region is device dependent. Currently, all I/O devices connected

to PTARM all interface with PTARM through single cycle memory mapped control registers. So memory instructions accessing I/O regions currently also take only one thread cycle.

Load/store instructions in ARM have the ability to update the base register after any memory operation. This compacts code that reads arrays, as a load or store instruction can access memory and updates the base register so the next memory access is done on the updated base register. The addressing mode of the instruction dictates how the base address register is updated. Pre-indexed addressing mode calculates the memory address by first using the value of the base register and offset, then updates the base register after the memory operation. Post-indexed addressing mode first updates the base register, then uses the updated base register value along with the offset to form the memory address. Offset addressing mode simply calculates the address from the base register and offset, and does not update the base register. When pre and post-indexed addressing modes are used, load operations require an additional thread cycle to complete. This results from the contention of the single write port in the register file. Because the register only has one write port, we cannot simultaneously write back a loaded result and update the base register in the same cycle. Thus, an extra pass through the pipeline is required to resolve the contention and update the base register.

**Load/Store Multiple**

The load/store multiple instruction is used to load (or store) a subset, or possibly all, of the general purpose registers from (or to) memory. This instruction is often used to compact code that pushes (or pops) registers to (or from) the program stack. The list of registers used is encoded in a 16 bit bit-vector as part of the instruction. The 0th bit of the bit-vector represents R0 and the 15th bit represents R15. A base register supplies the base memory address that is loaded from or stored to. The base address is sequentially incremented or decremented by 4 bytes and used as the memory address for each register that is subsequently operated on. Figure 3.10 shows how the load/store multiple instruction executes in the pipeline.

The load/store multiple instruction is inherently a multi-cycle instruction, because each thread cycle can only write back one value to the register or store one value to memory. When the instruction is initially decoded, the register list is read and stored in the thread state keep track of the instruction progress. During each execution cycle, the *register address decoder* in the pipeline decodes the register list and determines the register being operated on. For loads, this indicates the destination register that is written back to. For stores, this indicates the register whose value will be stored to memory. The *load/store multiple offset* block calculates the current memory address offset based on the remaining bits in the register list. The offset is added to the base register to form the memory address fed into memory. Each cycle, the register that is operated on is cleared from the remaining register list. The instruction completes execution when all registers have been operated on, which occurs when all bits in the register list are cleared.

The execution time of this instruction depends on the number of registers specified in the register list and the memory region that is being accessed. For accesses to the scratchpad or boot code, each register load or store takes only a single cycle. However, if memory accesses are to the DRAM region, then each register load/store will take multiple cycles. Load/store multiple instruction can also update the base register after all the register operations completes. Similar to the load/store register instruction, an additional thread cycle will be used to update the base register for load multiple instructions. Although the execution time of this instruction seems to be dynamic depending on the number of registers specified in the register list, but this number can be

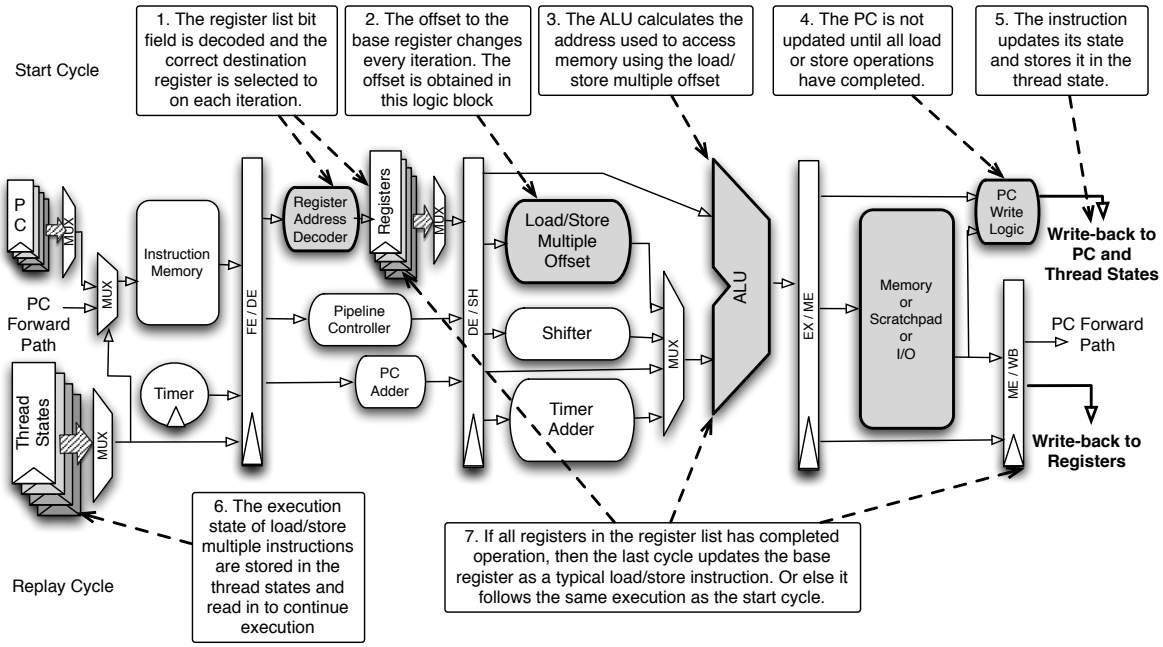



Figure 3.10: Load/Store Multiple Instruction Execution in the PTARM Pipeline

determined statically from the instruction binary. Thus, the execution time of this instruction can easily be statically analyzed.

### Load to PC

When a load instruction loads to R15, a branch is triggered in the pipeline. This is also the case for load multiple instructions when bit 15 is set in the register list bit-vector. In our five stage pipeline, the PC is updated in the memory stage to prepare for the next instruction fetch for the thread. However, if the branch target address is loaded from memory, the address is not yet present in the memory stage to be committed; only at the writeback stage will it be present. Thus, we introduce a forwarding path that forwards the PC straight from the writeback stage to instruction fetch if the next PC comes from memory. Figure 3.11 shows how this is implemented in our pipeline.

An extra multiplexer is placed in the fetch stage before the instruction fetch to select the forward path. When a load to R15 is detected, it will signal the thread state to use the forwarded PC on the next instruction fetch, instead of the one stored in next PC. We showed in figure 3.2 that for the same hardware thread, the fetch and writeback stage overlap in execution. As the memory load will be completed by the writeback stage, the correct branch target address will be selected and used in the fetch stage.

Section 2.1.1 discussed the timing implications of data-forwarding logic in the pipeline. Although it seems the selection of PC is dynamic, but when forwarding occurs is actually static; the PC forwarding only and always occur when instructions load from memory to R15. This mechanism has no additional timing effects on any following instructions, as no stalls are needed to wait for the address to be ready. Even if the load to R15 instruction is accessing the DRAM region, the

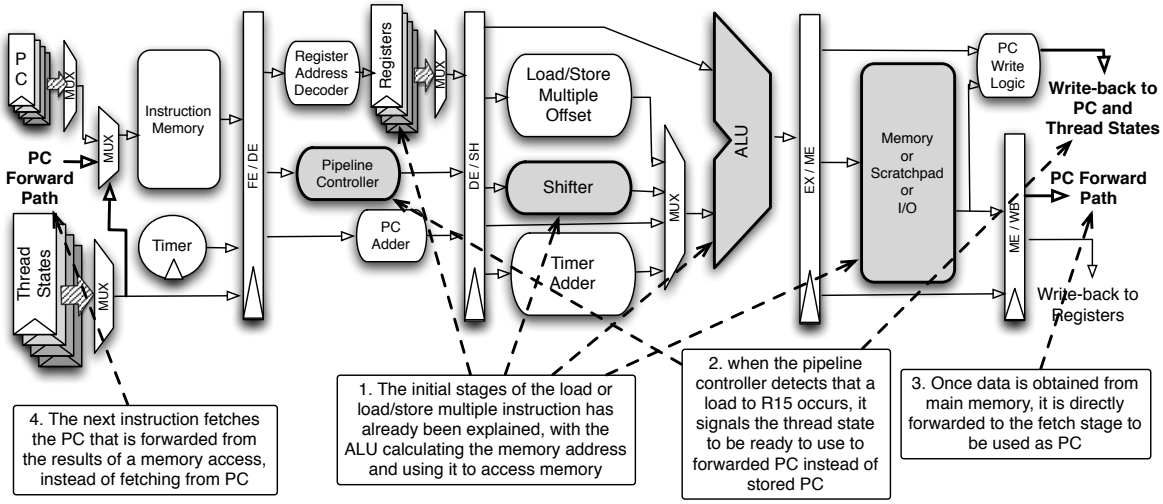



Figure 3.11: Load to R15 Instruction Execution in the PTARM Pipeline

execution time of this instruction does not deviate from a load instruction destined for other registers. Although the target address will not be known until after the DRAM access completes, a typical load instruction also waits until the DRAM access completes before the thread fetches the next instruction. So this extra forwarding mechanism does not cause load to R15 instructions to deviate from other load timing behaviors.

If the load to R15 instruction updates the base register, then the forwarding path is not needed and not used. The extra cycle used to update the base register will allow us to propagate the results from memory to update the PC in the memory stage. This timing behavior conforms to a typical load instruction that updates its base register.

### 3.4.4 Timing Instructions

Section 2.3 presented the instruction extensions to the ARM ISA to bring timing semantics at the ISA level. These instructions are added using the co-processor instruction slots in the ARM instruction space. In particular, the timing instructions are implemented using co-processor 13. Table 3.2 summarizes the instructions, their op codes, and their operations. All instructions have the assembly syntax "***cdp, p13,*** <***opcode***> ***rd, rn, rm, 0***", with <opcode> differentiating the instruction type.

All timing instructions use the *platform clock* to obtain and compare deadlines. Instead of using an external timer that is accessed through the I/O bus, the *platform clock* is implemented as a core hardware unit in the pipeline. The deterministic single cycle access latency to the clock value increases the precision of and predictability of timing operations on our processor. The *platform clock* is implemented in the *timer* hardware block shown in figure 3.1. An unsigned 64 bit value represents time in nanoseconds, and resets at zero when PTARM is reset. Unsigned 64 bits of nanoseconds covers approximately 584 years. The *platform clock* is implemented with a simple 64 bit adder increments to the current time value each processor clock cycle. We clock PTARM at $100MHz$, so the timer value is incremented by 10 nanoseconds every processor cycle. If the

| Type | Opcode | Functionality |
|---|---|---|
| ***get_time*** | 8 | timestamp = *current_time*;<br>crd = high32(timestamp);<br>crd+1 = low64(timestamp); |
| ***delay_until*** | 4 | deadline = (crm << 32) + crn;<br>while ( *current_time* < *deadline* )<br>  stall_thread(); |
| ***exception_on_expired*** | 2 | offset = (crm << 32) + crn;<br>register_exception(offset); |
| ***deactivate_exception*** | 3 | deactivate_exception(); |

Table 3.2: List of assembly deadline instructions

processor clock speed is modified, then the timer increment must be modified to reflect the correct clock speed. For architectures that allow the processor frequency to be scaled, the *platform clock* must also be adjusted when the frequency is scaled. For the purposes of clock synchronization, the time increment is stored in a programmable register that can adjust the timer increment to synchronize with external clocks. The timer increment value can only be modified through a privileged *set_time_increment* instruction, to protect the programmer from accidentally speeding up or slowing down the *platform clock*.

The timestamp associated with each instruction execution is latched during the fetch stage of the pipeline. In order words, the *time of execution* for each instruction is the precise moment when the instruction begins execution in the pipeline. Timestamps are 64 bits, so they require two 32 bit registers to store. The timestamps are loaded into general purpose registers with the *get_time* instruction, so standard register-to-register instructions can be used to manipulate the timestamps. PTARM does not currently provide 64 bit arithmetic operations, so programmers must handle the arithmetic overflow in software. The timing effects from the timing instructions are thread specific. Each thread operates on its own timestamps, and are not affected by the timing instructions from other threads. With 4 hardware threads interleaved through the pipeline, each hardware thread observes the time change once every 4 processor clock cycles. So the minimum observable interval of time for our implementation is $40ns$. The timing implications of this is discussed in section 3.6. We now describe how the pipeline executes each timing instruction.

### Get_Time

The *get_time* instruction is used to obtain the current clock value. The timestamp obtained from *get_time* represents the *time of execution* of this instruction. The execution of *get_time* is straightforward and shown in figure 3.12. The timestamp is latched during instruction fetch, and stored into registers. Because the register file only contains one write port, so *get_time* takes two thread cycles to complete; each cycle writes back 32 bits of the timestamp. The timestamp is written back to the destination register rd and rd+1, with rd storing the lower 32 bits and rd+1 storing the higher 32 bits. This instruction will not write to R15 (PC), and it will not cause a branch. If R14 or R15 is specified as rd, causing a potential write to R15, then this instruction will simply act as a NOP.

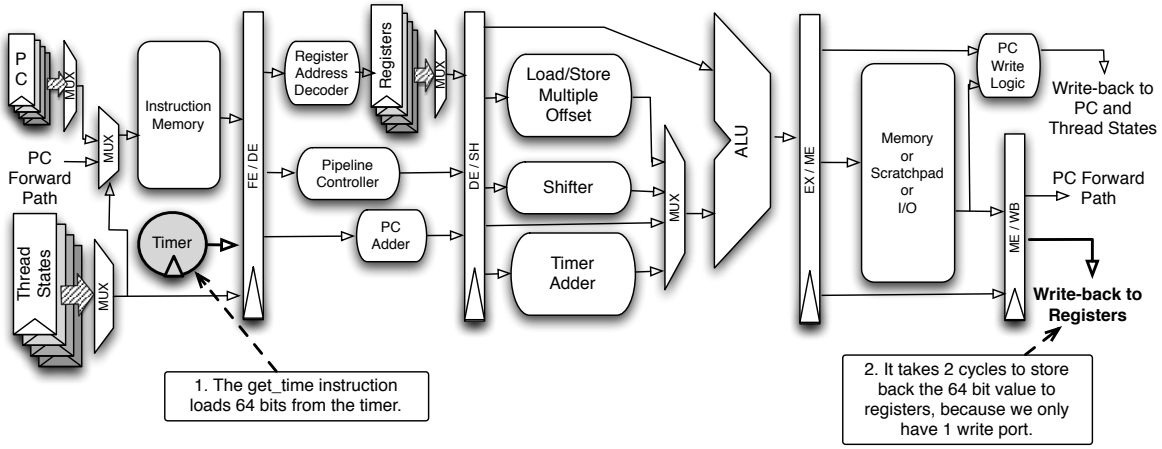



Figure 3.12: Get_Time Instruction Execution in the PTARM Pipeline

### Delay_Until

*Delay_until* is used to delay the execution of a thread until the *platform clock* exceeds an input timestamp. It takes in 2 source operands that forms the 64 bit timestamp that is checked against the *platform clock* every thread cycle. As described in section 2.3, the *delay_until* instruction can be used to specify a lower bound execution time for code blocks. This could be useful for synchronization between tasks or communicating with external devices. Figure 3.13 shows the execution of the *delay_until* instruction in the PTARM pipeline.

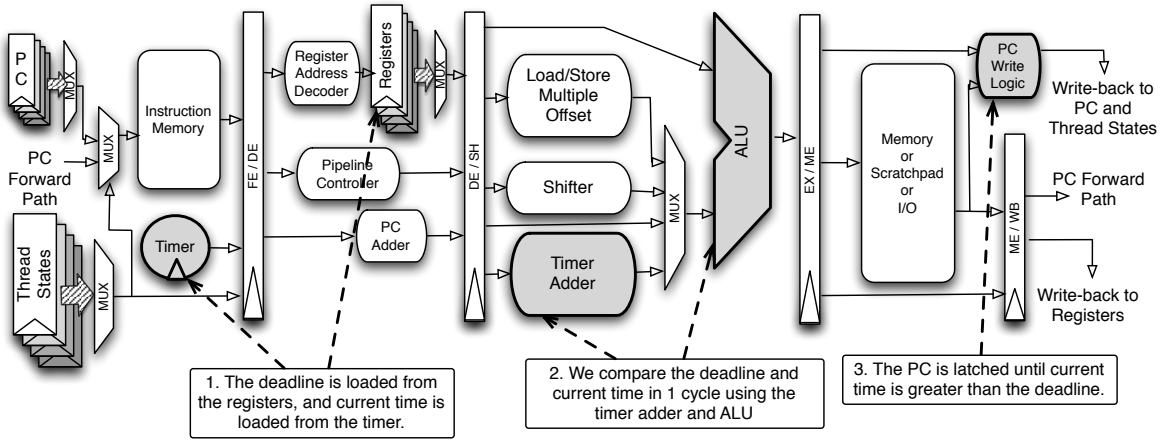



Figure 3.13: Delay_Until Instruction Execution in the PTARM Pipeline

The *delay_until* instruction highlights the reason *timer adder* is added into the pipeline. During the execution of *delay_until* the clock value is used every thread cycle to be compared with the input timestamp. However, the input timestamp and clock value are both 64 bit values. Without the additional *timer adder* in the pipeline, comparing 64 bits would require two thread cycles using our 32 bit ALU. This increases the jitter of this instruction by a factor of two, because

now the two timestamps can only be compared every two thread cycles. The added *timer adder* allows *delay_until* to compare the timestamps every thread cycle, and ensure that no additional threads cycles elapses after the input timestamp is reached. To delay program execution, the PC is only updated when the clock value is greater or equal to the input timestamp. No thread states are modified by *delay_until*. If the clock value already exceeds the input timestamp when the instruction is first decoded, then this instruction acts as a NOP. The PC is simply updated and the program execution continues. We detail the jitter effects of *delay_until* in section 3.6.2.

**Exception_on_Expire and Deactivate_Exception**

*Delay_until* passively compares an input timestamp against the platform clock when the instruction is executed. *Exception_on_expire* registers a timestamp to be actively checked against the *platform clock* in hardware. When the *platform clock* exceeds the registered timestamp value, a *timer_expired* exception is thrown. *Deactivate_exception* deactivates the timestamp that is actively being checked so no exception will be thrown. The idea is similar to setting of timer interrupts on embedded platforms, which are typically controlled through memory mapped registers.

Within the *timer* unit, there is one 64 bit deadline slot for each thread to register a timestamp to be actively checked. PTARM has 4 hardware threads, so there are four deadline slots in the *timer* unit. Whenever an *exception_on_expire* instruction is executed, the two source operands form the timestamp that is stored to the thread's corresponding deadline slot. The *exception_on_expire* instruction takes only one thread cycle to execute. It simply stores and activates the timestamp in the thread's deadline slot. Once activated, program execution continues, and the deadline slot timestamp is compared against the *platform clock* every thread cycle in the *timer* unit, until deactivated with *deactivate_exception*. When the *platform clock* is greater or equal to the stored timestamp, a *timer_expired* exception is triggered by the *timer* unit, and the deadline slot is deactivated to ensure only one exception is thrown per timestamp. When *deactivate_exception* is executed, if the deadline slot for the thread is active, then it will be deactivated. If the deadline slot for the thread is non active, then *deactivate_exception* will do nothing. The implementation of the *timer* unit is shown in figure 3.14.

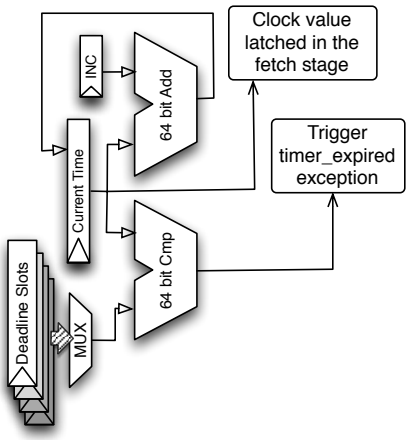


Figure 3.14: Implementation of Timer Unit

*Exception_on_expire* and *deactivate_exception* instructions are thread specific; each thread has its own dedicated deadline slot. The handling *timer_expired* exceptions, described in section 3.3, preserves temporal isolation for the hardware threads in the pipeline. So the timing effects of *exception_on_expire* and *deactivate_exception* can only affect the specific thread they are executed on. The timing details and jitter introduced with this mechanism is detailed in section 3.6.2.

Each thread currently can only check for one timestamp in hardware. To create the effects of multiple timestamps to being checked in hardware, the timestamps need to managed in software and share the one physical deadline slot. It is possible to add more deadline slots for threads in the *timer* unit at the cost of increased hardware usage. One deadline slot for each thread (4 deadline slots total) requires a multiplexer and a 64 bit comparator against the current clock, as shown

in figure 3.14. So more deadline slots would add more comparators and multiplexers, plus an additional OR gate to OR the exception triggering signal. The instructions *exception_on_expire* and *deactivate_exception* can easily be modified to take an id representing a specific deadline slot.

## 3.5 Implementations

### 3.5.1 PTARM VHDL Soft Core

The PTARM soft core is written in VHDL. It includes the pipeline, scratchpad memories, predictable memory controller and connects to several I/O devices on the FPGA. These include several LEDS, the UART, DVI controller and the DDRII DRAM. All I/O devices are connected through the I/O bus, while the DDRII DRAM is connected directly to the DRAM controller. Figure 3.15 shows the high level block diagram of the PTARM soft core.

The LEDs are memory mapped and can be toggled by setting and clearing bits. PTARM communicates to the UART through the UART gateway, which queues read and write requests from the core and relays it to the UART. The default buffer size on the UART gateway is one. The UART gateway status registers are mapped to memory I/O locations so programs can poll them to determine that status of the UART. Currently all read and write operations to the UART is done through blocking procedure calls. The UART runs at a baud rate of 115200 and can send and receive bytes.

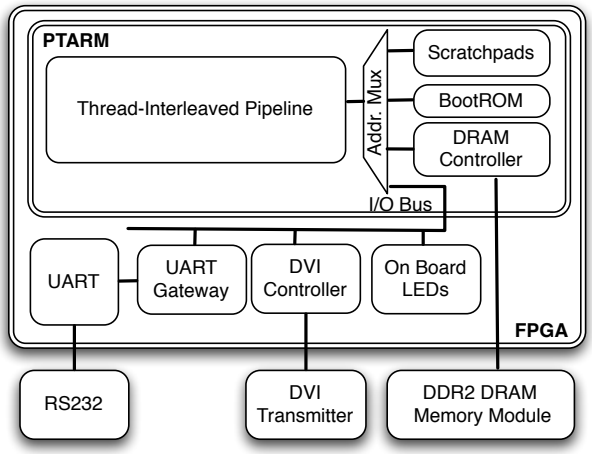



Figure 3.15: PTARM Block Level View

The DVI controller is used to control the DVI input port on the FPGA, so a monitor can be connected. We implemented a DVI controller application similar to the one presented in [45], where a vga controller is managed purely in software through the deadline instructions presented in the paper. Here, we use the timing constructs presented in section 2.3 to control the sending out of vertical and horizontal sync signals in software. As one hardware thread manages the sync signals, other hardware threads in our core can be used to calculate and draw to the screen buffer. Because hardware threads are temporally isolated, the timing of hardware sync signals are not affected by the operations on other hardware threads.

We synthesized the core on a Virtex-5 lx110t FPGA to obtain the maximum clock frequency and resource usage. (Todo: show area and clock speed on FPGA) We current clock PTARM at $100MHz$ and the memory controller at $200MHz$. All VHDL source code, software code samples, and instruction manual can be downloaded from http://chess.eecs.berkeley.edu/pret.

### 3.5.2 PTARM Simulator

Along with the VHDL soft core, we also developed a C++ cycle accurate software simulator of our architecture. The simulator is meant for architectural exploration, and provides better

debugging for software written for PTARM. The C++ simulator models the pipeline and memory hierarchy, including the predictable DRAM controller.

The simulator provides a framework that allows us to do more architectural exploration and observe the behavior. For example, the DMA units for each thread mentioned in section 3.2.3 that provide threads with the ability to transfer contents from the DRAM to the scratchpads in the background is currently only implemented on the simulator. The simulator allows us to quickly test out different software behaviors and verify our assumptions. The simulator and corresponding documentation can be downloaded from http://chess.eecs.berkeley.edu/pret.

(Todo: present benchmark numbers on the simulator and compare to others)

## 3.6 Timing Analysis

Worst-case execution time (WCET) analysis requires a combination of software analysis to determine the worst-case path, and architectural analysis to determine the time it takes to execute that paths on the underlying architecture. A plethora of research has been done on the software analysis of program paths. Wilhelm et al. [118] presented a survey of tools and techniques available for worst-case path enumeration and loop analysis etc. However, the precision of the WCET analysis of those techniques ultimately depend on the underlying architecture implementation [40]. Architectures that exhibit wildly unpredictable execution times will result in overly conservative WCET analysis, even if the software structure is simple. Designed as a predictable architecture, the instructions of PTARM all exhibit deterministic timing behaviors, allowing precise architectural analysis for the WCET analysis. Table 3.3 summarizes the execution time each instruction takes in terms of *thread cycles*.

| | | | **Memory Region Accessed** | |
|---|---|---|---|---|
| **Instruction** | **Latency** | **Instruction (*Addressing Mode*)** | **SPM/Boot** | **DRAM** |
| Data Processing | 1 | Load Register (*offset*) | 1 | $4^{\phi}$ |
| Branch | 1 | Load Register (*pre/post-indexed*) | 2 | $5^{\phi}$ |
| Software Interrupt (SWI) | 1 | Store Register (*all*) | 1 | $2^{\delta}$ |
| get_time | 2 | Load Multiple (*offset*) | $N_{reg}$ | $N_{reg} \times 4^{\phi}$ |
| delay_until | $1^{\dagger}$ | Load Multiple (*pre/post-indexed*) | $N_{reg} + 1$ | $(N_{reg} \times 4^{\phi}) + 1$ |
| exception_on_expire | 1 | Store Multiple (*all*) | $N_{reg}$ | $N_{reg} \times 2$ |
| deactivate_exception | 1 | | | |
| **Notes:** | | | | |
| $N_{reg}$: This is number of registers in the register list. | | | | |
| $^{\delta}$: The single store buffer (described in section 3.2.3) can hide the store latency to DRAM down a 1 thread cycle. But in cases where the store buffer cannot be used, the latency is 2 thread cycles. | | | | |
| $^{\phi}$: The dram load latency is 3 or 4 thread cycles depending on the alignment of the pipeline and the dram controller backend, as described in section 3.2.3. For conservative estimates, 4 thread cycles is used. | | | | |
| $^{\dagger}$: This is the minimum execution time of *delay_until*. The actual execution time varies depending on the input timestamp. | | | | |

Table 3.3: Timing properties of PTARM instructions (in thread cycles)

A *Thread cycle* is the unit used to represent execution time for each thread. Timing analysis can be done separately for each hardware thread running on PTARM because the threads are temporally isolated; the execution time of each thread is not affected by other threads. The thread-interleaved pipeline switches thread contexts every processor cycle in a predictable round

robin fashion. Thus, each thread is fetched and executed in the pipeline every $N$ processor cycles, $N$ being the number of threads in the pipeline. One *Thread cycle* represents each time the thread enters in the pipeline, which is the thread's perceived notion of cycles. The execution frequency of each thread ($F_{thread}$) is $F_{processor}/N$, so each *thread cycle* is $1/F_{thread}$ long. Our PTARM core is clocked at $100MHz$ ($F_{processor} = 100 \times 10^6$) and has 4 threads ($N = 4$) , so each thread cycle is $\frac{1}{(100\times10^6)/4} = 40 \times 10^{-9}$ secs, or $40ns$ long. The length of the *thread cycle* will not change because of the predictable thread-switching policy, making it a reliable unit of measurement for execution time.

### 3.6.1 Memory instructions

Data-processing and branch instructions have straightforward execution times. The execution time of branches are deterministic because the branch penalty is completely hidden by the thread interleaving. On the other hand, memory instructions in our architecture can have several different latencies depending on addressing mode or region of access, as listed in table 3.3. For memory instructions that use pre or post-indexed addressing mode to update the base register, an additional cycle latency is needed to write back to the base register. This is documented in the instruction implementation of load/store register in section 3.4.3. The addressing mode of load/store instructions is specified as part of the instruction binary. Thus, it can be determined statically and does not affect the complexity or precision of execution time analysis.

Different memory technologies provide different access latencies. The exposed memory hierarchy allows us to clearly label and identify access latencies based on the address accessed by the memory instruction. In execution time analysis tools, *value analysis* attempts to determine the address accessed by each instruction [118]. Once the *value analysis* determines the memory address, a precise memory access latency can be associated with the memory instruction. This allows for a simpler timing analysis and more accurate execution analysis compared to conventional memory hierarchies with caches. If caches are used to hide the memory hierarchy, additional modeling of the cache state is required after the *value analysis* to predict the cache state and determine if the access hit or missed the cache.

For store instructions, the single store buffer described in section 3.2.3 can usually hide the latency to access DRAM, if the subsequent instruction does not accesses the DRAM. Otherwise the store to DRAM will observe full memory access latency of two thread cycles. Architectural timing analysis can account for the store buffer by statically checking the next instructions to see if there are memory accessing instructions to the DRAM. The window of instructions that need to be checked is only one, so only slightly complicates the timing analysis. If it is not possible, then the full store to DRAM latency can be used for conservative analysis.

The execution time of load/store multiple instructions depend on the number of registers operated on, and the memory region it accesses. Because the register list is statically encoded in the instruction, the number of registers operated on can be determined statically. For each register that is operated on, the latency will depend on which memory region it accesses. The total execution time of the instruction will be the sum of the latencies for all register operations. Store multiple instructions to the DRAM do not benefit from the store buffer, because it issues consecutive stores to the DRAM. Thus, each store takes the full DRAM store latency. If pre or post-indexed addressing mode is used, an extra cycle is added to update the base register, just as regular load/store instructions.

### 3.6.2 Timing instructions

With the exception of *delay_until*, which by design exhibits variable execution time, the execution time of all other timing instructions are static. However, they can impact the execution time the program in a very dynamic way. For example, the execution of *exception_on_expire* and *deactivate_exception* only take one thread cycle, but when the *timer_expired* exception is thrown, the execution time of the whole program dynamically changed. To precisely understand the timing effects of the timing instructions, we must understand the jitter of the timing instructions caused by the underlying implementation. It is impossible for any hardware implementation to provide absolute precision of time, as we are limited by the digital synchronous circuits that discretize the notion of time. Although the timing extensions allow the manipulation of nanoseconds in software, with the thread-interleaved pipeline in PTARM, the basic unit of time for each thread is one thread cycle, or $40ns$. In other words, $40ns$ is the shortest interval of time that is observable by each thread. This can also be understood from the implementation of the thread-interleaved pipeline. Each thread only latches the clock value in the fetch stage, and the timestamp is propagated along the pipeline and associated with the instruction. Since there are four threads cycling in a round robin fashion, each thread latches the clock value only once every 4 processor cycles. With 100MHz clocking the pipeline in our implementation, 4 processor cycles is equivalent to $40ns$.

When manipulating timestamps, the execution time of the timing instructions and jitter must be accounted for. The timestamp associated with each instruction represent the *time of execution* of that instruction. In our implementation, the *time of execution* is when the instruction begins to execute, so the timestamp is latched in the fetch stage. This is the value store into registers for *get_time* instructions. Since *get_time* takes 2 thread cycles to complete, $80ns$ will have elapsed when the next instruction begins its execution. In the same way, *delay_until* delay programs execution until the current *time of execution* is $\geq$ the input timestamp value. When *delay_until* completes its execution, the next instruction will observe a the platform time of *at least* $40ns$ greater than the input timestamp passed to *delay_until*. This effect is illustrated in figure 3.16.

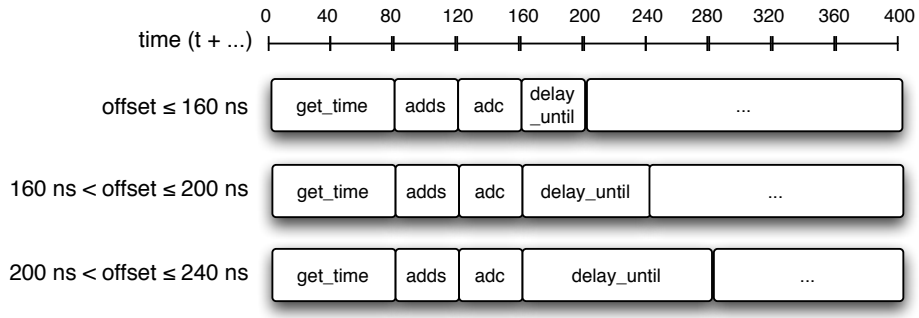



Figure 3.16: Timing details of get_time and delay_until

The code segment starts executing at time $t$. The code only consists of *get_time*, *delay_until*, and 2 add instructions used to add an offset to the timestamp obtained by *get_time*. In all 3 cases, the timestamp obtained by *get_time* would contain the value $t$, and the instruction after *get_time* executes at $t + 80$. Taking into account the 2 thread cycles used to add the offset to the timestamp, if the offset is $\leq 160$, then the *delay_until* will simply serve as a NOP. This is because

when *delay_until* is executed, it will latch $t + 160$ for the current time, and it will only delay program execution if the input timestamp is $> t + 160$. This is the top case shown in the figure. The instruction after *delay_until* executes at time $t + 200$, which accounts for the 1 thread cycle it takes to execute *delay_until*. Assuming *delay_until* does delay the program, in the worst-case, the instruction after *delay_until* can execute $79ns$ after the input timestamp. This can be observed if the offset is set to 161, which is shown in the middle timeline in figure 3.16. *Delay_until* will first latch the time $t + 160$ to compare with the input timestamp of $t + 161$. Because current platform time is less than the input timestamp, even by $1ns$, *delay_until* will delay the execution of the program until the next cycle, when $t + 200$ is latched to be compared against the input timestamp. At that point, *delay_until* will complete its execution, and the next instruction will execute at $t + 240$. This jitter results from the minimum observable time interval of $40ns$ for each thread, causing *delay_until* to have an observable jitter of up to $39ns$.

For each thread, the hardware *timer* unit checks an activated deadline slot once every thread cycle ($40ns$). Thus, the triggering of *timer_expired* exceptions from the *timer* unit also observes a similar jitter effect. This is illustrated in figure 3.17. If the thread has a deadline of

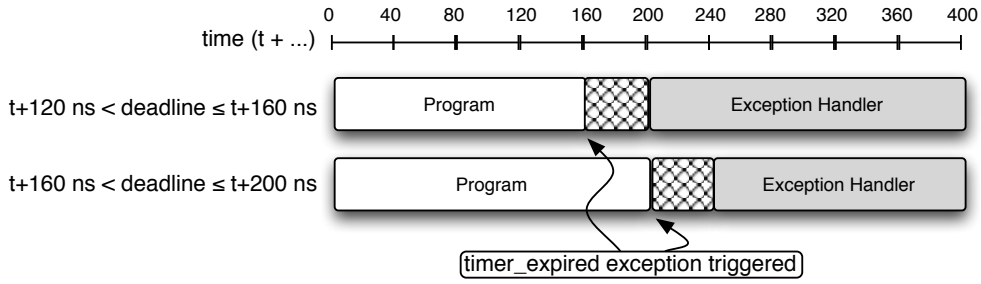



Figure 3.17: Timing details of the *timer_expired* exception triggering

$t + 161ns$, then the actual exception will not be triggered until $t + 200ns$, when the observed platform time is greater than the deadline.

### 3.6.3 Timed Loop revisited

We give a concrete example of analysis of timing instructions on PTARM by deriving the *offset* from the self compensating timed loop introduced in section 2.3.2. This timed loop detects whether the previous loop iteration missed its deadline. If it did, then the current iteration will execute a shorter version of the task in attempt to make up for the lost time, as shown in figure 3.18.

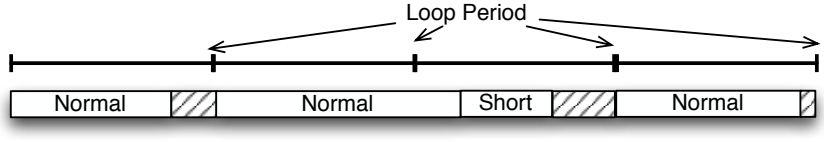



Figure 3.18: Execution of the self compensating timed loop

Listing 3.1: Timed loops with compensation revisited

```
1   cdp p13, 8, c2, c0, c0, 0  ; get_time, deadline timestamp stored in [c2, c3]
2 loop:
3   cdp p13, 8, c4, c0, c0, 0  ; get_time, current timestamp stored in [c4, c5]
4   subs r5, r5, #80           ; compensate for loop overhead and delay_until
5   sbc  r4, r4, #0            ;
6
7   subs r3, r3, r5            ; Check if previous iteration deadline is missed
8   sbc  r2, r2, r4            ;
9
10  blmi task_short            ; execute shorter task if previous deadline mess
11  blpl task_normal           ; or else execute normal task
12
13  adds r3, r3, #4000         ; assuming the deadline is 4 us (4000 ns)
14  adc r2, r2, #0             ; calculate the deadline timestamp for this iter.
15  cdp p13, 4, c2, c2, c3, 0  ; delay_until
16
17  b loop
```

### Obtaining the offset

Listing 3.1 shows the source code that is used to construct this timed loop. During the miss detection (lines 3 to 8), an additional *offset* is used to compensate for the execution of *delay_until* and loop overhead. Time elapses between the *delay_until* of the previous loop iteration (line 15), where the previous deadline timestamp is checked, and the *get_time* used for miss detection (line 3) in the current iteration. Without the offset compensation, the loop overhead will cause the miss detection to always detect a missed deadline. This can be observed from table 3.4, where we show a sample execution trace of four iterations in this timed loop. Figure 3.18 shows the timing behavior of these four iterations, where a missed deadline in the second iteration will cause the third iteration to compensate by executing the shorter version of the task.

In table 3.4, execution starts at time $t$. As mentioned before, each thread cycle is $40ns$, which is reflect in the left most column that shows the progression of time. We also show the thread cycle (TC) count, which starts at $n$ when execution begins. The execution time of each instruction is according to table 3.3. All instructions are compiled on the instruction scratchpad. In this code segment, we keep track of two timestamps each iteration. The *deadline_timestamp* keeps track of the loop deadlines, and is stored in registers r2 and r3. The *current_timestamp* is updated with *get_time* in the beginning of each loop iteration to detect if the previous iteration missed its deadline. It is stored in registers r4 and r5. The loop period is set to be $4\mu s$, which is $4000ns$ (100 thread cycles). We add the loop period to the *deadline_timestamp* in each loop iteration (lines 13 and 14).

The need for the *offset* can be observed at the beginning of the second loop iteration. At time $t + 4080ns$, *get_time* is called to initiate the miss detection sequence. The previous *deadline_timestamp* is $t + 4000$, which was met in the first iteration. However, *get_time* updates the *current_timestamp* to $t + 4080$, because the execution of *delay_until* and *b loop* took 2 thread cycles combined. Thus, our miss detection needs to account for this by subtracting the 2 thread cycles ($80ns$) difference from *current_timestamp* before comparing it with *deadline_timestamp*. In general, the *offset* that needs to be account for is the time elapsed between the deadline checking *delay_until* instruction and the miss detection *get_time* instruction. Intuitively, we want to check whether the previous *delay_until* executed before the previous *deadline_timestamp*, so the *offset* is calculates the

| Time | TC | Instruction | Comment |
|---|---|---|---|
| t ns | n | *cdp p13, 8, c2, c0, c0, 0* | get_time (deadline: t) |
| – – Loop 1st iteration / No deadline miss – – | | | |
| t+80 ns | n+2 | *cdp p13, 8, c4, c0, c0, 0* | get_time, (current: t+80) |
| t+160 ns | n+4 | *subs r5, r5, #80* | (current -= 80) |
| t+200 ns | n+5 | *sbc r2, r2, r4* | (current: t) |
| t+240 ns | n+6 | *subs r3, r3, r5* | compare deadline (t) and current (t) |
| t+280 ns | n+7 | *sbc r2, r2, r4* | result is 0, clear cc[“n”] |
| t+320 ns | n+8 | *blmi task_short* | nop since cc[“n”] == 0 |
| t+360 ns | n+9 | *blpl task_normal* | branch since cc[“n”] == 0 |
| - ns | - | . . . | executing task_normal |
| t+3800 ns | n+95 | *adds r3, r3, #4000* | (deadline += 4000) |
| t+3840 ns | n+96 | *adc r2, r2, #0* | (deadline: t+4000) |
| t+3880 ns | n+97 | *cdp p13, 4, c2, c2, c3, 0* | delay_until, input timestamp is t+4000 |
| - ns | - | . . . | delay_until for 3 thread cycles |
| t+4040 ns | n+101 | *b loop* | jump back to loop |
| – – Loop 2nd iteration / Deadline miss – – | | | |
| t+4080 ns | n+102 | *cdp p13, 8, c4, c0, c0, 0* | get_time, (current: t+4080) |
| t+4160 ns | n+104 | *subs r5, r5, #80* | (current -= 80) |
| t+4200 ns | n+105 | *sbc r2, r2, r4* | (current: t+4000) |
| t+4240 ns | n+106 | *subs r3, r3, r5* | compare deadline (t+4000) and current (t+4000) |
| t+4280 ns | n+107 | *sbc r2, r2, r4* | result is 0, clear cc[“n”] |
| t+4320 ns | n+108 | *blmi task_short* | nop since cc[“n”] == 0 |
| t+4360 ns | n+109 | *blpl task_normal* | branch since cc[“n”] == 0 |
| - ns | - | . . . | code for task_normal |
| t+7960 ns | n+199 | *adds r3, r3, #4000* | (deadline += 4000) |
| t+8000 ns | n+200 | *adc r2, r2, #0* | (deadline: t+8000) |
| t+8040 ns | n+201 | *cdp p13, 4, c2, c2, c3, 0* | delay_until, *no delay* |
| t+8080 ns | n+202 | *b loop* | jump back to loop |
| – – Loop 3rd iteration / Compensate with shorter task – – | | | |
| t+8120 ns | n+203 | *cdp p13, 8, c4, c0, c0, 0* | get_time, (current: t+8120) |
| t+8200 ns | n+205 | *subs r3, r3, r5* | (current -= 80) |
| t+8240 ns | n+206 | *sbc r2, r2, r4* | (current: t+8040) |
| t+8280 ns | n+207 | *subs r3, r3, r5* | compare deadline (t+8000) and current (t+8040) |
| t+8320 ns | n+208 | *sbc r2, r2, r4* | result is -40, set cc[“n”] |
| t+8360 ns | n+209 | *blmi task_short* | branch since cc[“n”] == 1 |
| - ns | - | . . . | code for task_short |
| t+10280 ns | n+257 | *blpl task_normal* | nop since cc[“n”] == 1 |
| t+10320 ns | n+258 | *adds r3, r3, #4000* | (deadline += 4000) |
| t+10360 ns | n+259 | *adc r2, r2, #0* | (deadline: t+12000) |
| t+10400 ns | n+260 | *cdp p13, 4, c2, c2, c3, 0* | delay_until |
| - ns | - | . . . | delay until time is t+12000 |
| t+12040 ns | n+301 | *b loop* | jump back to loop |
| – – Loop 4th iteration / Execute normal task – – | | | |
| t+12080 ns | n+302 | *cdp p13, 8, c4, c0, c0, 0* | get_time, (current: t+12080) |
| t+12160 ns | n+304 | *subs r3, r3, r5* | (current -= 80) |
| t+12200 ns | n+305 | *sbc r2, r2, r4* | (current: t+12000) |
| t+12240 ns | n+306 | *subs r3, r3, r5* | compare deadline (t+12000) and current (t+12000) |
| t+12280 ns | n+307 | *sbc r2, r2, r4* | result is 0, clear cc[“n”] |

Table 3.4: Instruction execution trace of the self compensating timed loop
(TC = thread cycles)

time of execution of the previous *delay_until*.

**Overhead of the self compensating timed loop**

In this self compensating timed loop, the loop period is set to $4000ns$ and regulated with the *delay_until* instruction. Each loop period includes the execution of the actual task along with the loop and timing control overhead. The loop overhead in this example is only the branch instruction on line 17 in listing 3.1, which is 1 thread cycle (40 ns). The overhead for timing control and self compensation consists of all the timing instructions, the arithmetic on the timestamps, and the 2 conditional branch instructions that determines which task to execute. From table 3.4 we can count a total overhead of 11 thread cycles which includes: 1 *get_time* (2 thread cycles), 1 *delay_until* (1 thread cycle), 6 arithmetic operations on the timestamps (6 thread cycles), and 2 conditional branch instructions (2 thread cycles). Overall the timed loop contains an overhead of 12 thread cycles ($480ns$) , which means both tasks have a soft timing requirement of 88 thread cycles ($3520ns$) for each loop iteration to meet its deadline. In the second loop iteration of our example, *task_normal* executed for 89 thread cycles, exactly one thread cycle over its timing requirement. As a result, the *delay_until* of the second loop iteration did not delay program execution, and the third iteration miss detection detects a missed deadline, and switches to execute *task_short*.

**First loop iteration jitter**

The *offset* previously derived is the time difference between the desired deadline and the time of execution of the *get_time* used for miss detection. In our code, because of the simple loop structure, the *offset* only included the execution time of the *delay_until* and a branch. However, if the difference was larger, for example, in a conditional loop structure, then it could introduce jitter for the first iteration. An example is shown in figure 3.19. We assume that the setup code remains

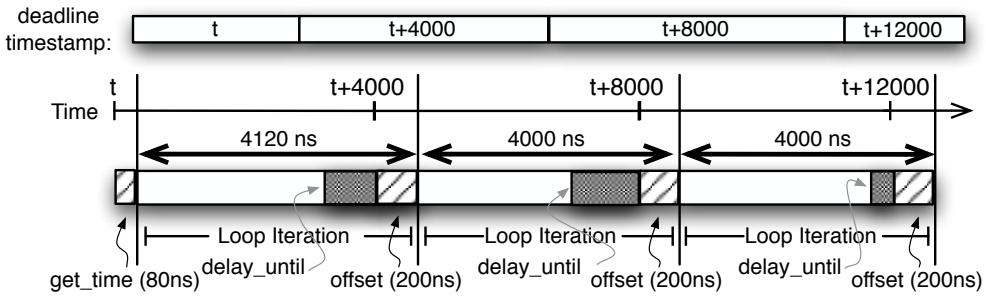



Figure 3.19: Jitter caused by initial timed loop setup

the same with only one *get_time*, and the *offset* is adjusted to 200ns. We also assume that the loop period remains the same, 4000ns, and all loop iterations meet the loop period timing requirements. In this example, we see that the first iteration executes for 120ns longer than subsequent iterations. The jitter introduced for the first iteration is exactly the execution time difference between the *offset* and the setup code. Between each *delay_until* instruction, exactly 4000 ns elapses, since 4000 ns is used as the loop period and added to the *deadline_timestamp* each loop iteration. From the figure we observe that the execution of *offset* occurs between each subsequent *delay_until* instruction. However, between the initial *deadline_timestamp* value ($t$) and the first *delay_until*, the only overhead

that is observed is the execution of a *get_time* instruction, which is 80 ns. Thus, the first iteration of the loop executes for an addition $120ns$, which is the difference between the *offset* and the execution time of the loop setup code. This effect is not observed in the previous example because the execution time of both *offset* and loop setup is $80ns$, so the first iteration also executed for exactly 4000 ns.

Listing 3.2: Jitter adjusted timed loop

```
1   mov r6, #0                  ; i = 0;
2   mov r7, #0                  ; j = 0;
3
4   cdp p13, 8, c2, c0, c0, 0   ; get_time, deadline timestamp stored in [c2, c3]
5   subs r3, r3, #40            ; adjustment for first loop period
6   sbc  r2, r2, #0             ; deadline -= 40
7 loop:
8   cdp p13, 8, c4, c0, c0, 0   ; get_time, current timestamp stored in [c4, c5]
9   subs r5, r5, #200           ; compensate for loop overhead and delay_until
10  sbc  r4, r4, #0             ;
11
12  subs r3, r3, r5             ; Check if previous iteration deadline is missed
13  sbc  r2, r2, r4             ;
14
15  blmi task_short             ; execute shorter task if previous deadline mess
16  blpl task_normal            ; or else execute normal task
17
18  adds r3, r3, #4000          ; assuming the deadline is 4 us (4000 ns)
19  adc r2, r2, #0              ; calculate the deadline timestamp for this iter.
20  cdp p13, 4, c2, c2, c3, 0   ; delay_until
21
22  add r6, r6, #1              ; i += 1
23  add r7, r7, r6 LSL #1       ; j += i*2
24  cmp r7, #1000               ;
25  blt loop                    ; branch back if ( j < 10000 )
```

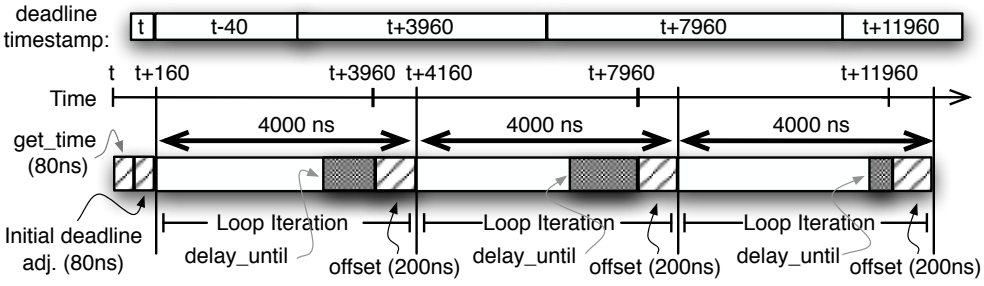



Figure 3.20: Adjusted timed loop setup

This first iteration jitter can be accounted for by adjusting the initial *deadline_timestamp* in the loop setup code. In listing 3.2 we show the source code that adjusts for the initial loop iteration jitter. Lines 22 to 25 show the additional loop overhead that conditionally checks whether to branch back to the beginning of the loop. The *offset* that we have to adjust for in this example is exactly 200ns, which includes the 4 instructions for the loop overhead and the *delay_until*. This offset is accounted for on line 9. Lines 4 to 6 show the loop setup code where we adjust for the execution time of the initial loop iteration. 40 is subtracted from the initial *deadline_timestamp* obtained by

the *get_time* on line 4. This value is obtained by the execution time difference of the *offset* ($200ns$) and the setup code ($160ns$). We show the resulting timing behavior in figure 3.20, where the first loop iteration is adjusted to 4000ns, the same as subsequent iterations. By entering the loop with the *deadline_timestamp* value of $t - 40$, we shift the *delay_until* deadlines for all loop iterations by 40 ns, which compensates for the initial loop iteration jitter. Intuitively, the initial *deadline_timestamp* is adjusted before the loop to create the illusion that the setup code and the loop overhead observed between each *delay_until* has the same execution time. By doing so, the first loop iteration will observe the same loop period as all subsequent iterations.

### 3.6.4 Exceptions

In section 3.3 we described how exceptions are thrown in PTARM. When an exception is triggered in one hardware thread, none of the other hardware threads are affected, as the pipeline is not flushed. For the thread on which the exception occurred, only one thread cycle is lost, and the control flow jumps to the correct exception handler depending on the exception vector table. Here, we give a concrete example of the timing behavior of exceptions on PTARM by using *exception_on_expire* and *deactivate_exception* to trigger a *timer_expired* exception. The code for the example is shown in listing 3.3.

Listing 3.3: Sample code that triggers a *timer_expired* exception

```
1   mov  r3, #0x98              ; r3 = address of _timer_handler_loc
2   add  r4, pc, #32            ; r4 = addr of delay_handler
3   str  r4, [r3]               ; register delay_handler
4
5   cdp  p13, 8, c2, c0, c0, 0  ; get_time
6   adds r3, #240
7   adc  r2, #0
8   cdp  p13, 2, c2, c2, c3, 0  ; exception_on_expire
9
10  add  r5, r6, r6             ; arbitrary code block
11  add  r7, r5, r6             ;
12
13  cdp  p13, 5, c8, c2, c3, 0  ; deactivate_exception
14  b end_program
15
16 delay_handler:
17  mov pc, lr                  ; simply return
```

In this example, a *delay_handler* (lines 16 and 17) is implemented to simply return when called. The *delay_handler* is registered as the user-level exception handler (lines 1 to 3) for the *timer_expired* exception. This is done by deriving the address of the *delay_handler* on line 2, and storing it to the *_timer_handler_loc*. The *_timer_handler_loc* is a reserved location that points to the address of a handler executed when the *timer_expired* exception is thrown. When a *timer_expired* exception occurs, the current address is saved and control flow jumps to the exception table entry for the *timer_expired* exception. This table entry redirects execution to a *timer_expired* exception setup code which calls the user-level exception handler registered. This setup code is shown in listing 3.4. The setup code loads the address of *_timer_handler_loc* into a register, and jumps to the handler on line 7. If the handler returns, lines 9 to 11 re-enable interrupts, and line 12 returns control to the original PC when the exception occurred.

Listing 3.4: The *timer_expired* exception setup code

```
1  .text
2  .global _tmr_exp_setup;
3  _tmr_exp_setup:
4      push  {r0, lr}                ; push registers to stack
5      ldr   r0, _timer_handler_loc  ; load address of timer expired exception handler
6      mov   lr, pc                  ; get return address after calling handler
7      mov   pc, r0                  ; jump to exception handler
8
9      mrs   r0, cpsr                ; get CPSR
10     bic   r0, r0, #0x80           ; enable interrupts
11     msr   cpsr, r0                ; write to CPSR
12     pop   {r0, pc}                ; pop stack and return from exception
13
14 _timer_handler_loc: .word  0x00000000;
```

The execution trace of this example is shown in table 3.5. Because execution jumps back and forth between the main code, the *timer_expired* setup code, and the *delay_handler*, we show the address of the instructions to help follow which code segment is being executed. The user code is compiled to start at 0x40000000, which is in the instruction scratchpad space. As described in section 3.2, the exception vector table and *timer_expired* setup code are all compiled as part of the boot code. The *str* instruction is storing to the *_timer_handler_loc*, which is a reserved location in the boot code, so executes in one thread cycle. The deadline timestamp is set so the *timer_expired* exception is thrown during the execution of the code block, which occurs at time $t + 360$. Although the address of execution at that time is 0x400000020, the instruction at that address does not complete, because the *timer_expired* exception is thrown in that thread cycle. That address is saved to the *link register* (R14) by the hardware when the exception is thrown. The next thread cycle, the exception vector entry for the *timer_expired* entry (at address 0x1C) is executed. The entry forces a branch to the *timer_expired* setup code, which executes to call *delay_handler*. The *push* and *pop* instructions are load/store multiple instructions that operate on the stack, compiled to the data scratchpad. Because these instructions are operating on 2 registers each, so they take at least 2 thread cycles to access the data scratchpad. In addition, they both update the base stack register, so *pop*, which loads from memory to the registers, takes an additional cycle to complete.

In section 3.3 we discussed the potential execution variability for exception handling if the instruction interrupted by the exception is accessing the DRAM. In order to maintain a deterministic execution time, we must ensure that the first 3 thread cycles (the worst-case DRAM access latency) after an exception is thrown does not access the DRAM. The exposed memory hierarchy with scratchpads allows us to statically compile the exception setup code and data stack, both accessed right after an exception is thrown, onto the scratchpad. This ensures that the DRAM is not accessed during the first 3 thread cycles after the exception is thrown, and allows for predictable exception handling.

We show that the timing analysis of exceptions is deterministic and straightforward in the PTARM architecture. No flushing of the pipeline occurs, no other hardware threads are affected, and the hardware exception throwing mechanism only observes a single thread cycle overhead. Due to deterministic instruction execution time and exposed memory hierarchy, the *response time* of hardware exceptions, which is the time elapsed between when the exception is thrown and when the user registered exception handler executes, is deterministic and can be statically obtained. For the

| Time | TC | Address | Inst | Comment |
|---|---|---|---|---|
| t ns | n | 0x40000000 | *mov r3, #0x98* | gets the *_timer_handler_loc* |
| t+40 ns | n+1 | 0x40000004 | *add r4, pc, #32* | get *delay_handler* address |
| t+80 ns | n+2 | 0x40000008 | *str r4, [r3]* | register *delay_handler* as timer expire handler |
| t+120 ns | n+3 | 0x40000014 | *cdp p13, 8, c2, c0, c0, 0* | get_time (timestamp: t+120) |
| t+200 ns | n+5 | 0x400000C | *adds r3, #240* | timestamp += 240 |
| t+240 ns | n+6 | 0x40000010 | *adc r2, #0* | timestamp: t+360 |
| t+280 ns | n+7 | 0x40000018 | *cdp p13, 2, c2, c2, c3, 0* | exception_on_expire, input timestamp: t+360 |
| t+320 ns | n+8 | 0x4000001C | *add r5, r6, r6* | code block |
| t+360 ns | n+9 | 0x40000020 | **throw exception** | timer expired, hardware exception thrown |
| t+400 ns | n+10 | 0x1C | *b _tmr_exp_setup* | branch to setup code |
| t+440 ns | n+11 | 0x78 | *push* {*r0, lr*} | push registers to stack |
| t+520 ns | n+13 | 0x7C | *ldr r0, _timer_handler_loc* | load address of timer expired handler |
| t+560 ns | n+14 | 0x80 | *mov lr, pc* | store return address after timer handler |
| t+600 ns | n+15 | 0x84 | *mov pc, r0* | jump to handler (*delay_handler*) |
| t+640 ns | n+16 | 0x4000002C | *mov pc, lr* | *delay_handler* code, return |
| t+680 ns | n+17 | 0x88 | *mrs r0, cpsr* | get CPSR |
| t+720 ns | n+18 | 0x8C | *bic r0, r0, #0x80* | enable interrupts |
| t+760 ns | n+19 | 0x90 | *msr cpsr, r0* | write to CPSR |
| t+800 ns | n+20 | 0x94 | *pop* {*r0, pc*} | pop stack and return from exception |
| t+920 ns | n+23 | 0x40000020 | *add r7, r5, r6* | re-execute instruction |
| t+960 ns | n+24 | 0x40000024 | *cdp p13, 3, c2, c0, c1, 0* | *deactivate_exception* (does nothing) |
| t+1000 ns | n+25 | 0x40000028 | *b end_program* | jump to end of program |

Table 3.5: Exception_on_expire sample code timing details

*timer_expired* exception in PTARM, the response time is 8 thread cycles ($320ns$), which includes the one thread cycle when the exception is thrown, and 7 thread cycles for code executed from the exception vector table and *timer_expired* setup code. This is reflected in table 3.5.

# Chapter 4

# Applications

In this chapter we will present two applications that have been implemented with our Precision Timed Architecture. The first application is a real-time one dimensional computational fluid dynamics (1D-CFD) simulator. This simulator runs in real-time to simulate the fuel rail pressure and flow rate for improved engine efficiently when injecting fuel. The application makes use of the light weight hardware threads in our thread-interleaved pipeline to implement a massively parallel simulator with hundreds of computational nodes communication to its neighbors. The timing predictable architecture allows us to statically analyze the execution time for each node to ensure that the execution time for each computational node can meet the timing constraints imposed by the application. A timed base communication scheme is implemented to reduce communication overhead. The communication synchronization is enforced in software with timing instructions to minimize overhead and enforce that communication occurs on-time and all nodes are in sync. We present the synthesis results on a Xilinx Virtex-6 FPGA to show that we can successfully simulate a common fuel rail configuration of up to 234 nodes.

The second application shows how we use our predictable architecture to eliminate timing side-channel attacks for encryption algorithms. Time-exploiting attacks take advantage of variations in execution time of cryptosystems to deduce the encryption keys. The root cause of these time-exploiting attacks is the uncontrollable run-time variance that is caused by the underlying architecture, allowing attackers to bypass the strong mathematical properties of the encryption and deduce the keys. We show that by using a timing-predictable architecture that provides more control of execution time to the programmer, we remove the vulnerability that is used to initiate the attack, and remove architecture deficiencies that can lead to more timing-attacks. We demonstrate this by running RSA and DSA encryption algorithms on PRET, which successfully illustrates the use of PRETs timing-centric methods to counter time-exploiting attacks.

## 4.1 Real-Time 1D Computational Fluid Dynamics Simulator

Modern diesel engines inject diesel fuel with high pressure into the combustion chamber for combustion. A digital control value is used to control the amount of fuel injected, which depends on the pressure and fuel rate of the fuel rails delivering the fuel. Several pilot injections are injected ahead of the main injection to mitigate the inject delay in the chamber and reduce audible noise. However, these pilot injections send pulsations through the fuel supply rail that need to be modeled

or damped before subsequent injection events to ensure the correct amount of fuel is injected [17]. Currently, fuel rails are modeled and developed with 1D-CFD solvers like GT-Fuel, and use an ad-hoc model of fuel pressure for injection events [120]. 1D-CFD models are commonly used in simulating transient operation of internal combustion engines [99]. Here, we present an implementation for real-time execution of a 1D-CFD solver using multiple PRET cores that model the fuel rail. Although the calculations are slightly rougher than the GT-Fuel calculations, it is sufficient to allow improved fuel pressure estimation and close the loop of fuel delivery, allowing for a cleaner, more efficient engine.

### 4.1.1 Background

The 1D CFD model of the fuel rail system is descried as a network of pipes. The system is built up from different types of pipe segments, which each model the fluid dynamics of a segment in the fuel rail. A fixed time step solver is implemented. At each time step, the pipe segments calculate its current pressure and flow-rate, and communicate these to its neighboring pipe segments to be used in the next time step. The time step is determined by the speed of information flow that is expressed in equation 4.1.

$$\frac{\Delta t}{\Delta x} a = C \tag{4.1}$$

In this equation, $a$ is the wave speed, $C$ is the Courant number and $\Delta x$ is the descretization length. For stability, the Courant number needs to be less than 1 and a number below 0.8 is recommended [35]. For example, if a fluid has a wave speed $a$ of 1 $cm$ per microsecond and a discretization length $\Delta x$ of 1 $cm$, then we require a time step $\Delta t$ of less than one microsecond. This discretization length of a pipe network is dominated by its smallest sub-volume and a 1 $cm$ discretization length is common for diesel fuel systems. For diesel engines, a speed of sound (wave speed) of 1500 $m/s$ [103] is commonly used. The real-time requirements of this application thus require adequate performance so that the slowest node can complete in $\Delta t$. Although highly parallel, the heterogeneity of pipe elements differentiates this application from typical homogeneous parallel problems often solved using GPUs or SIMD with large common memories [125], such as in image processing applications.

In order to evaluate our system of pipes we define a few types of computing nodes that corresponding to different pipe elements. These are shown in table 4.1 with derived pressure and flow rate equations. From these pipe elements we can generate a network of pipes that represent

| **Type** | (***Pressure***) $P_{I_n} =$ | (***Flow Rate***) $Q_{I_n} =$ |
|---|---|---|
| Pipe Segment | $\frac{(C_P + C_M)}{2}$ | $\frac{(P_{I_n} + C_m)}{B}$ |
| Imposed pressure | $P_{Bnd}$ | $\frac{(P_{Bnd} - C_M)}{B}$ |
| Imposed mass flow | $C_M + BQ_{Bnd}$ | $Q_{Bnd}$ |
| Valve | $C_P - BQ_{I_n}$ | $\frac{-BC_V + \sqrt{(BC_V)^2 + 2C_V C_P}}{C_V = \frac{(Q_0 \tau)^2}{2P_0}}$ |
| Cap | $C_P - BQ_{I_n}$ | 0 |
| "T" intersection | $\frac{\frac{C_{P_1}}{B_1} + \frac{C_{M_2}}{B_2} + \frac{C_{M_3}}{B_3}}{\sum \frac{1}{B}}$ | $-\frac{P_I}{B_1} + \frac{C_{P_1}}{B_1}$<br>$-\frac{P_I}{B_2} + \frac{C_{M_2}}{B_2}$<br>$-\frac{P_I}{B_3} + \frac{C_{M_3}}{B_3}$ |

Table 4.1: Table of supported pipe elements and their derived equations

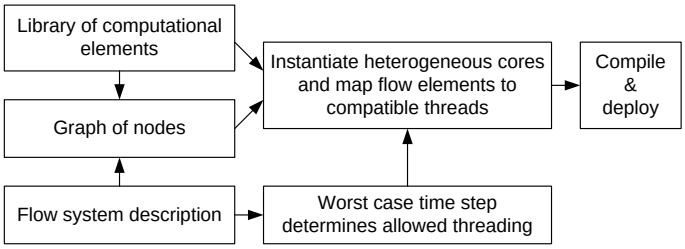



Figure 4.1: Design Flow

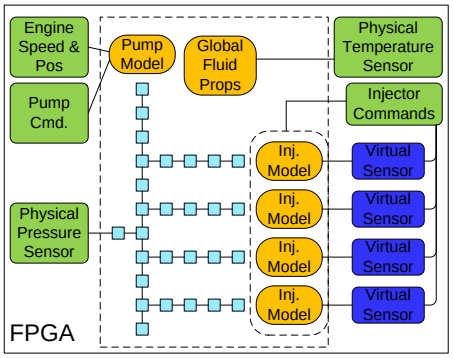



Figure 4.2: High Level System Diagram

our fuel system. The *imposed pressure* is used to represent the pressure sensor on the fuel system. The *imposed mass flow* is used to represent pump, and the *valve* is typically used to represent an injector. *Pipe segments* and *pipe "T"* are the interconnected pipe elements, and the *cap* is used to represent the end of a pipe. The derived equations shown in the table use the following simplified characteristic equations derived in [109].

$$C_P = P_{i-1} + Q_{i-1}\left(B - R|Q_{i-1}|\right) \text{ and} \tag{4.2}$$

$$C_M = P_{i+1} - Q_{i+1}\left(B - R|Q_{i+1}|\right). \tag{4.3}$$

In the equations, $B = a\rho/A$ and $R = \rho f \Delta x/2DA^2$, where $A$ is the cross sectional area of the pipe, and $Q$ is the flow rate along the pipe. $P$ is pressure, $\rho$ is fluid density, $V$ fluid velocity, $f$ is the Darcy-Weisbach friction factor, $D$ is pipe diameter, and $a$ is the wave speed. The $_{Bnd}$ subscript denotes a boundary condition. $C_v$ is the flow coefficient which is a function of: $Q_0$ the nominal open flow, $P_0$ the downstream pressure, and $\tau$ the fraction the valve is open. The $_{i+1}$ subscript and $_{i-1}$ subscript represent values that are received from the neighboring pipe elements. Any implementation of the system must ensure that these calculations for all pipe elements can be completed within the specified time step.

Figure 4.2 shows an overview of a representative system for modeling fuel rails. The 1D-CFD model is bounded inside the dashed rectangle. External to that is the real-word sensor and actuator interfaces that provide boundary conditions or consume model output variables. The small blue squares inside the dashed rectangle represent the network of pipes. In a practical simulation of a diesel fuel system the total number of pipe elements can range from around 50 to a few hundred. The overall design flow of generating the 1D-CFD model is shown in figure 4.1. The flow system description describes the fuel rail configuration, which is used to create a graph that describes the system, and determine the system parameters and time step requirements. With the graph and library of elements, we instantiate the hardware implementation, then compile and deploy the system.

For illustrative purposes, we show a sample pipe network graph in figure 4.3. Each pipe element is also referred to as a computational node. Its graphical representation is shown in Table 4.4. This pipe network starts with an imposed flow input (P1) element on the left, which represents a pump. Fluid travels through a few pipe segment nodes (P2 and P3) to a "T" intersection (P4), where it splits off to a second branch of the network. The "T" node is also measured by the outside world (D1) through a output port. Output elements are used when data needs to be communicated out of the model to other parts of the FPGA. Flow going up the new leg ends in a cap (P8), while flow

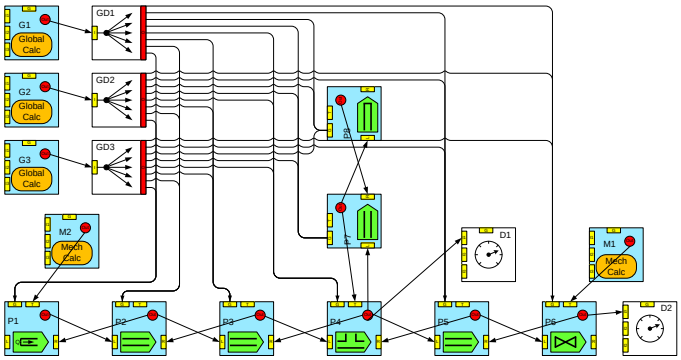



Figure 4.3: Detailed System Diagram

| | | | |
|---|---|---|---|
| Pipe segment | | Cap | |
| Imposed pressure | | Imposed flow | |
| Pipe "T" | | Valve | |
| Mechanical calculation | | Global calculation | |
| Global distribution | | Output | |

Figure 4.4: Library of Computational Node Elements

continuing down the original path exits the system through a valve (P6). *Mechanical calculation* elements compute the inputs to valve, defined flow, and defined pressure blocks. The system is assumed to be at uniform temperature. Temperature dependent variables like density and wave speed are computed by the global calculation nodes (G1, G2, and G3). This values are needed by all computational elements in the graph, thus are distributed by the global distributions (GD1, GD2, and GD3) to each of the computational elements every time step.

### 4.1.2 Implementation

This application presents several requirements that must be considered when being implemented. First, the whole system operates in time steps, which serves as the timing constraints that the longest executed computation node must meet. Second, communication is exchanged between nodes only once each time step, so synchronization is required between the heterogeneous nodes that exhibit varying execution times. Third, a typical fuel rail configuration range from fifty to several hundred pipe elements, thus any implementation must be scalable enough to support the larger configurations. With these requirements in mind, we will detail the implementation of the 1D-CFD simulator with Precision Timed Architectures.

#### Hardware Architecture

**PTARM Cores** Our hardware implementation synthesizes multiple PTARM cores connected through point-to-point connections on an FPGA. Computational nodes are each mapped to hardware threads on the PTARM cores. The PTARM core used for this application is a slightly modified version of the one presented in chapter 3. In order to improve the throughput and clock frequency of our pipeline, we implemented a six-stage thread-interleaved pipeline shown in figure 4.5. This thread-interleaved pipeline follows the same design principles as discussed in chapter 2, and supports a minimum of six threads interleaved through the pipeline. The memory footprint for each of the computational nodes range from roughly 100 to 1000 bytes. Thus, scratchpad memories are sufficient to hold all instructions and data for all threads within a PTARM core, no external memory is required. The pipeline also contains hardware floating point units to support the applications needs of floating point computations. The floating point units are single-precision, and generated using the Xilinx

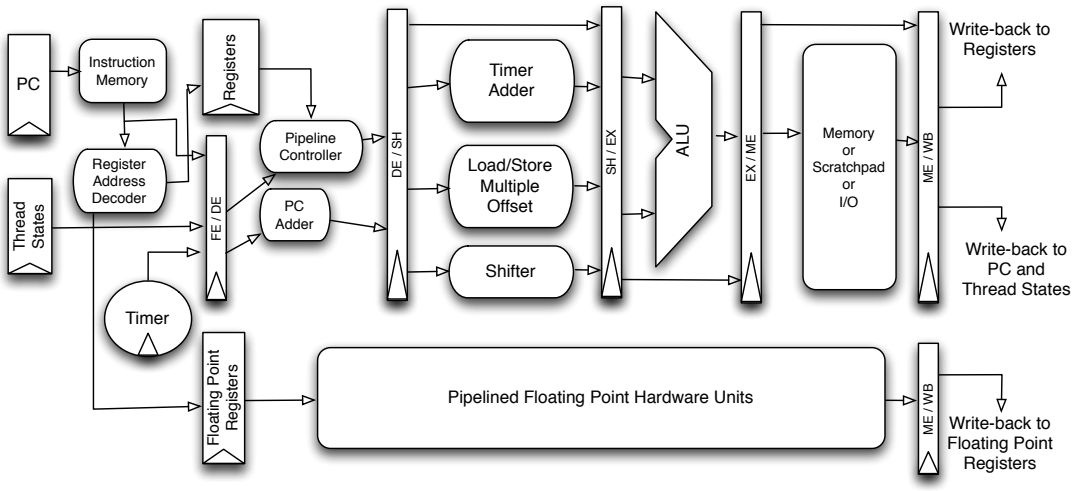



Figure 4.5: The PTARM 6 Stage Pipeline

Coregen tool [122]. They are pipelined to accept input every cycle, which avoids structural hazards, as explained in section 2.1.3. The floating point operations supported are: add, subtract, multiply, float-to-fix, fix-to-float, divide and square root.

Our pipeline design supports configurations which exclude certain floating point units, since not all computational nodes require all floating operations. For example, square root is only used by the valve node, and divide is only used by the "T" node, as shown in table 4.2. The floating point divide and square root hardware are the most resource intensive units, but the valve and "T" nodes usually represent only a few percent of overall system. The common fuel rail system we will present later contains 234 nodes, but only 5 nodes are "T" nodes and only 4 nodes are valves. To save on hardware resources, we could use software emulation for the complex operations at the cost of increase the execution time of the "T" nodes and valve nodes. However, the overall performance of our system is bounded by the slowest computational element, because all nodes synchronize communication points at the end of each time step. As a result, the performance hit from using software emulation for these small percent of nodes would limit the overall performance. Instead, by allowing different configurations of PTARM cores within the system, we can include the hardware additions only on cores that require them, getting the performance boost from hardware without a huge resource overhead. This results in substantial resource savings, which we show in section 4.1.3.

The real-time, highly parallel yet heterogeneous nature of this application makes it a perfect match for our Precision Timed Architecture. As explained in section 2.1.3, thread-interleaved pipelines contain simpler pipeline architectures, allowing for higher clock frequencies and less resource usage. The sharing of the data-path between multiple hardware threads further allows us to optimize the resource usage per computational element. The thread-interleaved pipeline also maximizes throughput over latency, which benefits this highly parallel application. The pipeline hide the latencies of multi-cycle operations, such as floating point operations, with execution from other threads. E.g., in our implementation, the normally 4 processor cycle floating-point additions and subtractions appear as single thread cycle instructions because their latencies are fully hidden by the thread interleaving.

**Interconnect** This application requires support for two types of communication. Between neighboring nodes, the pressure and flow rate values computed are exchanged every time step. Across the system, several temperature dependent parameters are calculated and broadcast to all nodes every time step as well. Thus, along with point to point communications between nodes, we also implement a global broadcast circuit. Each node can receive up to four inputs and transmit four outputs each time step, depending on the number of neighboring nodes it is connected to. Out of the inputs, one is dedicated to receiving broadcasts from the global distribution circuit.

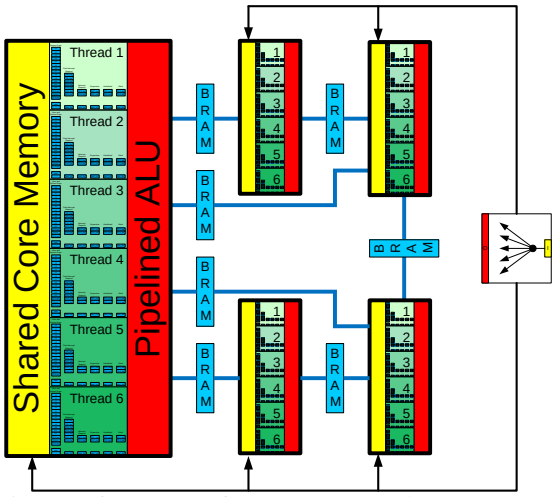


Figure 4.6: System of PRET Cores and Interconnects

Because nodes are mapped to hardware threads on a core, their neighboring node may be mapped to another thread on the same core, or a thread on a neighboring core. Nodes mapped to the same core (intra-core communication) communicate through the shared scratchpad memory within the core. For nodes mapped to different cores (inter-core communication), we use privately shared Block RAMs (BRAMs) between cores to establish the point-to-point communication channel. BRAMs are dedicated memories on the FPGA that provide single cycle deterministic access latencies, scratchpad memories within each core are also synthesized to BRAMs. Because the communication bandwidth requirements are small, we only need one shared BRAM between two cores to establish communication channels for all threads on both cores. This allows all threads to communicate with each other with single cycle latency, whether it is intra-core or inter-core communication. As an added benefit, by using BRAMs for communication, we save the logic slices on the FPGA to implement more cores to support bigger models. On modern FPGAs, the limiting resource factor is typically logic slices, not BRAMs. Each core only requires a small number of BRAMs to be used for registers and scratchpads, so the BRAM utilization ratio is far less than the logic slice utilization ratio when we synthesize many cores. As we present our synthesis results in section 4.1.3, we will show that the number of cores synthesized is indeed limited by the logic slices, not the BRAMs.

When implementing the global distribution circuit, we observed that only a few nodes are required to the broadcast all the temperature dependent parameters. In fact, in diesel fuel systems, the number of nodes needed to broadcast all parameters can be mapped to the six threads of one single PTARM core. Thus, we dedicate one PTARM core in the system as the broadcast core. For each other core, we add a dedicated broadcast receiving memory that is connected to the broadcast core. The broadcast receiving memory is also synthesized to a small dual-port BRAM, with a read-only port connected to the core, and a write-only port connected to the broadcaster. The broadcast core contains a broadcast bus that can simultaneously write to all the broadcast memories the same values. The broadcast memory is also shared amongst all threads in a core so all threads can access the global values. This architecture allows us to save on the resources needed to implement a full fledged interconnect routing system or any network protocol to be used for broadcasting. Figure 4.6 shows a block-level view of the hardware architecture.

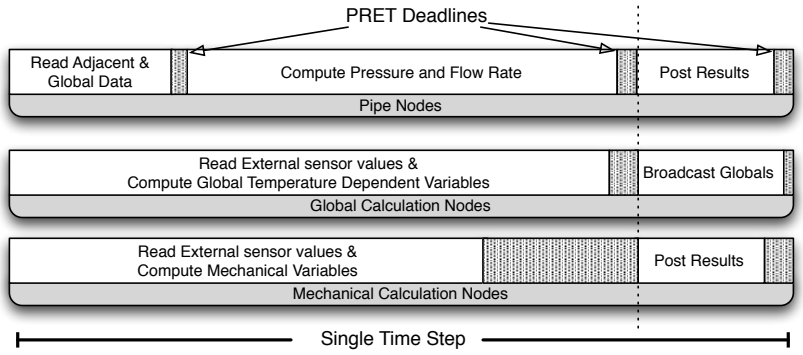



Figure 4.7: Execution of Nodes at Each Time Step

### Software Architecture

We implement the equations in table 4.1 in the language C and compile it with the GNU ARM cross compiler [3] to run on our cores. In order to minimize the computation required, the equations are statically optimized. The communication channels in and out of each node is memory mapped to the shared BRAMs between cores.

The execution of the system progresses in time steps. Computational nodes have varying execution speeds, to avoid data races and ensure all communication is synchronized each time step, we enforce an execution model where each time step consists of several synchronized phases, as shown in figure 4.7. For pipe nodes that read in neighboring data, shown on the top of figure 4.7, the first phase of each time step is to read in pressure and flow rate values from neighboring nodes, and the temperature dependent variables from the global broadcasters. Once input values are read, the computation occurs according to the specific fluid dynamics equations. The final phase of each time step, the computed results are posted to be used in the next time step. For global and mechanical nodes, the two phases consists of reading in external values for calculation, and posting results. We synchronize the data exchange between nodes to ensure avoid data races and ensure that all data operated on is consistent and from the same time step. This communication model is very similar to Giotto [42], where tasks communicate explicitly through ports, and only at the end of execution of the tasks to ensure deterministic communication between the tasks. While implementations of Giotto use an explicit run-time system to enforce the execution model, we use the timing instructions provided by the PRET architecture to implement our system.

In section 2.3 we introduced ISA extensions that provide programmers with explicit timing control in software. The implementation of the various timing instructions for PTARM is explained in section 3.4. Specifically for this application, we use a specialized timing macro *delay_and_set*, which uses *delay_until*, as introduced in section 2.3. The semantics of the *delay_and_set* instruction is similar to the deadline instruction introduced by Ip and Edwards [45]. When it is decoded, it first enforces the previously specified timing constraint, then it sets a new timing constraint for the next code block. The instruction enforces a minimum execution time within the code, which we use to enforce the synchronized execution of time steps for all nodes. Fig. 4.7 shows the program synchronization points that our timing instruction enforces. The hatched area in the figure denotes slack time that is generated by the timing instructions. Each *delay_and_set* instruction takes 2 thread cycles because it manipulates a 64-bit value representing time. For our computational nodes, 3 timing instructions are used each time step, thus 6 thread cycles of overhead are introduced per time

step.

The timing instructions provide a very lightweight and simple mechanism to enforce synchronization in software. No additional run-time system is needed to enforce the execution model, and we avoid the need to use locks or mutex to ensure correct ordering of communicated data. The same effect can possibly be achieved with no overhead using instruction counting and NOP insertions. This can certainly be done on any deterministic architecture such as PRET. However, NOP insertion is both brittle and tedious. Any change in the code would change the timing of the software and insertions need to be adjusted to ensure the correct number of NOPs are added. Designs now are mostly written in programming languages like the language C and compiled into assembly, making it even more difficult to gauge the number of NOPs needed at design time. The timing instructions allow for a much more scalable and flexible approach. In a system with heterogeneous nodes and different execution times, the timing instructions allow us to set the same timing constraints in all nodes regardless of its execution time.

The *delay_and_set* instruction only enforces minimum execution time, but does not guarantee that the worst-case execution time of all computational nodes meet the timing constraints imposed from the application. Static timing analysis on all nodes is still required to verify that the worst-case execution time of time steps meets the imposed timing constraints by the application parameters. However, as soon as the timing constraints are met, there are no additional benefits to improving the execution speed of the computational nodes. As system time steps are synchronized with sensors that interface with the physical world, and execution is real-time along the engine. In this case, precise execution time analysis can help us optimize other system resources, such as power and area, improving the scalability of the approach. On the other hand, over estimation of execution time could lead to over-provisioning of hardware resources. In this application, the computation code on the nodes within each time step contains only a single path of execution, voiding the need for complex software analysis. Thus, the predictability of the underlying architecture determines how precise the worst-case execution time analysis is. Communication is handled by the synchronized communication points, which enforces an ordering between the writing and reading of shared data. This voids the need of any explicit synchronization methods, removing any overhead and unpredictability for communication. The underlying architecture uses the time-predictable PRET, and implements a latency-deterministic communication network of shared BRAMs on the FPGA. These properties allow us to statically obtain an exact execution time for each computation node, which we will show and present in the next section.

### 4.1.3 Experimental Results and Discussion

We use three examples to evaluate our framework. The first example is a simple waterhammer example taken from Wylie and Streeter [121]. It is similar to the one shown in figure 4.3, but without the "T" element and the nodes that branch up. This example contains an imposed pressure, 5 pipe segments, a valve, and two mechanical input blocks that provide both the reference pressure and the valve angle as a function of time. We use this simply as a sanity check for the correctness of functionality of our framework.

The second and third example cover two common diesel injector configurations: the unit pump and common rail. The data for configuring these cases was taken from reference examples provided by Gamma Technologies' GT-SUITE software package [35]. The unit pump is much like the simple waterhammer case in that there are no branches in the system. The input is a defined flow

| | Without Interpolation / With Interpolation | | | | | |
|---|---|---|---|---|---|---|
| Type | Mul | Add/Sub | Abs | Sqrt | Div | Thread cycles |
| Pipe segment | 10 / 18 | 5 / 13 | 2 / 2 | 0 / 0 | 0 / 0 | 51 / 81 |
| Imposed pressure | 6 / 10 | 3 / 7 | 1 / 1 | 0 / 0 | 0 / 0 | 38 / 50 |
| Imposed flow | 5 / 9 | 3 / 7 | 1 / 1 | 0 / 0 | 0 / 0 | 40 / 51 |
| Valve | 13 / 17 | 5 / 9 | 1 / 1 | 1 / 1 | 0 / 0 | 55 / 64 |
| Cap | 4 / 8 | 2 / 6 | 1 / 1 | 0 / 0 | 0 / 0 | 39 / 48 |
| Pipe "T" | 16 / 28 | 13 / 25 | 3 / 0 | 0 / 0 | 4 / 4 | 72 / 111 |

Table 4.2: Computational Intensity of Supported Types

specified by an electronically controlled cam driven pump. The output is a single valve. There are a total of 73 fluid sub-volumes in this system. The common rail example is more complex where the topology is roughly that described by the 1D-CFD model in figure 4.3. It has a total of 234 sub-volumes, including 5 "T" intersections and 4 valves. Both the GT-SUITE-based models use a 1 $cm$ discretization length, which, using a 1500 $m/s$ wave speed, and a stability factor of 0.8 yields a 5.33$\mu s$ time step to complete our worst-case instructions for the slowest computational node.

We synthesize all our cores and interconnects on the Xilinx Virtex 6 xc6vlx195t [123] with speed grade 3. Each Virtex-6 FPGA logic slice contains 4 LUTs and 8 flip-flops, and this FPGA contains 31,200 logic slices and 512 18-$KB$ BRAMs. Each PRET core is clocked at 150 $MHz$ and has 6 threads. All floating point units are generated from the Xilinx Coregen tool [122], and are configured to maximize DSP slice usage and minimize logic slice usage as much as possible. We save the logic slices to synthesize as many cores as possible. Our current PRET implementation, the PTARM, uses an ARM-based ISA, thus our C code is compiled using the GNU ARM cross compiler [3] with the optimization compiler flag set to level 3. For these examples, we used a mapping heuristic that grouped nodes requiring same computations onto the same core. In the sections below we will show that this heuristic allows us to save hardware resources by synthesizing less floating point units.

**Timing Requirements Validation**

In order to ensure that the worst-case computational element can meet the timing requirements, static timing analysis is done on all computational nodes to determine the worst-case execution of each time step. As discussed in section 4.1.2, the computation code within each time step only consists of a single path, simplifying the timing analysis. The thread-interleaved pipeline provides temporal isolation for all hardware threads, so no timing interference occurs between the threads. We can safely use the timing analysis done separately for each computational node even as they are executed simultaneously in the architecture. Because all code, data, and communication channels reside on the BRAMs of the FPGA, the access latency is all deterministically one cycle. The PTARM architecture provides deterministic execution time for each instruction implemented, and the full list of instruction execution cycles is listed in table 3.3. Most floating point instructions take only a single thread cycle, as the latency is fully hidden by interleaving the hardware threads in the pipeline. The more complex floating point square root and divide operations take four thread cycles. Using the deterministic instruction execution cycles and the compiled code, we are able to obtain the exact thread cycles required for each computational node, which is shown in table 4.2.

To convert thread cycles to physical time, we use the processor clock speed and number of threads executing in the architecture. Given a 150 $MHz$ clock rate and six hardware threads, each thread executes at 25 $Mhz$ in our thread-interleaved pipeline. Thus, each thread cycle converted to physical time is 40 $ns$ long. The unit pump and common rail have a requirement of 5.33 $\mu s$, which gives us 133 thread cycles to complete the computation each time step. Table 4.2 shows that the "T" element, which takes 111 thread cycles with interpolation, is the node with the worst-case execution time, well below the 133 thread cycle constraint. For the simple waterhammer example, a bigger discretization $\Delta x$ is used, which leads to a bigger time step than that of the two complex examples. This validates that we can safely meet the timing requirements, ensuring the correctness of functionality of our implementation.

**Resource Utilization**

Table 4.3 shows the resource usage in logic slices for different configurations of a PTARM core. Each core uses 7 BRAMs: 3 for the integer unit register set (3 read and 1 write port), 2 for floating point register set (2 read and 1 write port), 1 for the scratchpad, and 1 for the global broadcast receiving memory. We include the fixed point configuration for reference purposes, as it doesn't contain any floating point units. The baseline configuration used in our implementation is the "basic float", which contains a floating point add/subtracter, a floating point multiplier, and float to fix conversion units. The "sqrt", "div" and "sqrt & div" configurations add the corresponding hardware units onto the "basic float" configuration. Besides the effect of hardware units, we also show the area impact of adjusting the thread count on a single core.

| Threads per core | 6 | 8 | 9 | 16 |
|---|---|---|---|---|
| Fixed point only | 572 | 588 | 764 | 779 |
| Basic float | 820 | 823 | 1000 | 1022 |
| Float with sqrt | 987 | 992 | 1146 | 1172 |
| Float with div | 1039 | 1051 | 1231 | 1237 |
| Float with div & sqrt | 1237 | 1249 | 1403 | 1413 |

Table 4.3: Number of Occupied Slices per Core on the Virtex 6 (xc6vlx195t) FPGA.

Two important observations are made from the results of table 4.3. First, the area increase associated with adding more threads to the core is proportional only to the number of bits required to encode the number of threads. For example, running 6 threads or 8 threads (both requiring three bits to encode the thread number) on the processor yields a similar area usage. But once a 9th thread is introduced, the used area noticeably increases, but remains similar for up to 16 threads. This can be explained by understanding the architecture of multi-threaded processors. Multi-threaded processors maintain independent register sets and processor states for each thread, while sharing the datapath and ALU units amongst all threads. The register sets are synthesized onto BRAMs, so the number of bits used to encode thread IDs will determine how big of a BRAM is used for the register set. The size of the multiplexers used to select thread states and registers is also determined by the number of bits encoding the thread IDs, not the actual number of threads running. Thus, it is possible to increase the number of threads per core with almost negligible impact on area as long as the incremented thread count uses the same number of bits to encode. Increasing the thread

capacities will allow our architecture to support more nodes in a single FPGA. However, since hardware threads share the processor pipeline, adding threads slows down the running speed of the individual threads. Nonetheless, for implementation that have sufficient slack time or require faster performance, adjusting the number of threads could lead to a valuable improvement. Our precise execution time analysis allows us to determine the maximum number of threads, six in our case, we can support to meet our timing constraints. An over estimated execution time in this case could lead to under utilizing the hardware by constraining the number of threads to five, resulting in requiring additional cores to implement our 237 node fuel rail example.

The second observation relates to the resource impact of the floating point square root and divide units. Looking at the resource usage for 6 threads on a core, adding a floating point square root unit adds roughly 20.3% more logic slices than the "basic float" configuration. Ading a floating point division unit adds roughly 26.7% more logic slices than the "basic float" configuration. A core with both square root and division unit would use roughly 50.8% more slices. These are estimates because the slices occupied might vary slightly based on how the synthesis tool maps LUTs and flip flops to logic slices. But they give an intuition to the resource difference used for each configuration.

The actual resource impact can be seen from Table 4.4, which shows the total slices occupied when the three examples we implemented are synthesized. In the homogeneous (hom. suffix) configuration, all the cores contain the square root and divide hardware. In the heterogeneous (het. suffix) configuration, only necessary cores contain square root and divide, the rest use the basic float configuration.

| Example | | Nodes | Cores / Conn. | Slices / BRAM | |
|---|---|---|---|---|---|
| | | | | Absolute | Relative (%) |
| Water Hammer | het. | 12 | 2 / 1 | 1805 / 15 | 5.7 / 2.1 |
| | hom. | | | 2379 / 15 | 7.6 / 2.1 |
| Unit Pump | het. | 73 | 13 / 12 | 10566 / 103 | 33.0 / 15.0 |
| | hom. | | | 16635 / 103 | 44.0 / 15.0 |
| Common Rail | het. | 234 | 39 / 38 | 29134 / 311 | 93.4 / 45.0 |
| | hom. | | | N/A | |

Table 4.4: Total Resource Utilization of Examples Synthesized on the Virtex 6 (xc6vlx195t) FPGA

For the simple waterhammer example, since only 2 cores are used, the savings is less noticeable. But as the application size scales up, the resource savings of a heterogeneous architecture become more apparent. The homogeneous approach uses roughly 1.5 times the number of slices our heterogeneous approach uses, which is consistent with the findings in table 4.3. This proved to be critical for the 234-node common rail example, as only our heterogeneous architecture could implement the design on the xc6vlx195t FPGA while the homogeneous design simply could not fit. These results also reflect our decision to use a heuristic that groups nodes with the similar computation together. By doing so, we can synthesize less hardware floating point units overall, saving hardware resources. Table 4.4 also shows the BRAM usage for the implemented examples. Each interconnect uses 1 BRAM and each core uses 7 BRAMs. We see that the BRAM utilization ratio is far below the logic cell utilization, validating our design choice of using BRAMs for interconnects and broadcasts.

### 4.1.4 Conclusion

In this application, we presented a novel framework for solving a class of heterogeneous micro-parallel problems. Specifically we showed that our approach is sufficient to model a diesel fuel system in real time using the 1D-CFD approach on FPGAs. To the best of our knowledge, we believe this is the first attempt to attack real-time CFD on this timescale and complexity of problem. There may exist different implementation options for our application on FPGAs. For example, we could attempt the problem in discrete FPGA blocks. However, in order to make the application fit in a practical FPGA, we would need to re-use the hardware multipliers, adders, and other functional units. This would require a state machine to run it and begins to look a great deal like a processor.

Instead, we use the PRET architecture to ensure timing determinism and implement a light-weight timing based synchronization on a multicore PRET architecture. We set up a configurable heterogeneous architecture that leverages the programmability of FPGAs to efficiently synthesize the design for efficient area usage. Our results show ample resource savings, proving that our approach is practical and scalable to larger and more complex systems.

## 4.2 Eliminating Timing Side-Channel-Attacks

Encryption algorithms are based on strong mathematical properties to prevent attackers from deciphering the encrypted content. However, their implementations in software naturally introduce varying run times because of data-dependent control flow paths. Timing attacks [52] exploit this variability in cryptosystems and extract additional information from executions of the cipher. These can lead to deciphering the secret key. Kocher describes a timing attack as a basic signal detection problem [52]. The "signal" is the timing variation caused by the key's bits when running the cipher, while "noise" is the measurement inaccuracy and timing variations from other factors such as architecture unpredictability and multitasking. This signal to noise ratio determines the number of samples required for the attack – the greater the "noise," the more difficult the attack. It was generally conceived that this "noise" effectively masked the "signal," thereby shielding encryption systems from timing attacks. However, practical implementations of the attack have since been presented [23, 29, 126] that clearly indicate the "noise" by itself is insufficient protection. In fact, the architectural unpredictability that was initially believed to prevent timing attacks was discovered to enable even more attacks. Computer architects use caches, branch predictors and complex pipelines to improve the average-case performance while keeping these optimizations invisible to the programmer. These enhancements, however, result in unpredictable and uncontrollable timing behaviors, which were all shown to be vulnerabilities that led to side-channel attacks [18, 81, 5, 28].

In order to not be confused with Kocher's [52] terminology of *timing attacks* on algorithmic timing differences, we classify all above attacks that exploit the timing variability of software implementation *or* hardware architectures as *time-exploiting attacks*. In our case, a *timing attack* is only one possible *time-exploiting attack*. Other time-exploiting attacks include branch predictor, and cache attacks. Examples of other side-channel attacks are power attacks [68, 51], fault injection attacks [20, 33], and many others [126].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [20, 28, 53, 48, 4, 49, 26, 112, 111]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-

channel exploits [28, 20, 53, 48, 4, 49, 26, 112, 111] indicates that we do not have the complete set of solutions as more and more vulnerabilities are still being discovered and exploited. Just recently, Coppens et al. [28] discovered two previously unknown time-exploiting attacks on modern x86 processors caused by the out-of-order execution and the variable latency instructions. This suggests that while current prevention methods are effective at *defending* against their particular attacks, they do not *prevent* other attacks from occurring. This, we believe, is because they do not address the root cause of time-exploiting attacks, which is that run time variability *cannot be controlled* by the programmer.

It is important to understand that the main reason for time-exploiting attacks is *not* that the program runs in a varying amount of time, but that this variability *cannot be controlled* by the programmer. The subtle difference is that if timing variability is introduced in a controlled manner, then it is still possible to control the timing information that is leaked during execution, which can be effective against time-exploiting attacks. However, because of the programmer's *lack of control* over these timing information leaks in modern architectures, noise injection techniques are widely adopted in attempt to make the attack infeasible. These include adding random delays [52] or blinding signatures [52, 26]. Other techniques such as branch equalization [73, 126] use software techniques to rewrite algorithms such that they take equal time to execute during each conditional branch. We take a different approach, and directly address the crux of the problem, which is the *lack of control* over timing behaviors in software. We propose the use of an embedded computer architecture that is designed to allow predictable and controllable timing behaviors.

At first it may seem that a predictable architecture makes the attacker's task simpler, because it reduces the amount of "noise" emitted from the underlying architecture. However, we contend that in order for timing behaviors to be controllable, the underlying architecture *must* be predictable. This is because it is meaningless to specify any timing semantics in software if the underlying architecture is unable to honor them. And in order to guarantee the execution of the timing specifications, the architecture must be predictable. Our approach does not attempt to increase the difficulty in performing time-exploiting attacks, but to eliminate them completely.

For this application, we present PRET in the context of embedded cryptosystems, and show that an architecture designed for predictability and controllability effectively eliminates all time-exploiting attacks. We target embedded applications such as smartcard readers [53], key-card gates [22], set-top boxes [53], and thumbpods [94], which are a good fit for PRET's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [75] encryption algorithms on PRET, and show its immunity against time-exploiting attacks. This work shows that a disciplined defense against time-exploiting attacks requires a combination of software and hardware techniques that ensure controllability and predictability.

### 4.2.1 Background

Kocher outlined a notion of timing attacks [52] on encryption algorithms such as RSA and DSS that require a large number of plaintext-ciphertext pairs and a detailed knowledge of the target implementation. By simulating the target system with predicted keys, and measuring the run time to perform the private key operations, the actual key could be derived one bit at a time. Kocher also introduced power attacks [68, 51], which use the varying power consumption of the processor to infer the activity of the encryption software over time. These played a large role in stimulating research in side-channel cryptanalysis [74, 49], which also found side-channel attacks

against IDEA, RC5 and blowfish [49]. Fault-based attacks [20, 48, 33] were introduced by Bihan et al. [20]. These attacks attempt to extract keys by observing the system behavior to generated faults. For the side-channel attacks that we have missed, Zhou [126] presents a survey on a wide range of side-channel attacks.

Dhem et al. [29] demonstrated a practical implementation of timing attacks on RSA for smart cards and the ability to obtain a 512-bit key in a reasonable amount of time. Several software solutions such as RSA blinding [52, 26], execution time padding [52], and adding random delays [52] have been proposed as possible defenses against this attack. However, these solutions were not widely adopted by the general public until Brumley et al. [23] orchestrated a successful timing attack over the local network on an OpenSSL-based web server. This motivated further research on timing attacks for other encryption algorithms such as ECC [33] and AES [18]. In particular, Bernstien's attack on AES [18] targeted the the run time variance of caches. The introduction of simultaneous multi-threading (SMT) architectures escalated this type of attack on shared hardware components. Percival [81] showed a different caching attack method on SMT, made possible because caches were shared by all processes running on the hardware architecture. Acimez et al. introduced branch predictor attacks [5, 4] that monitor control flow by occupying a shared branched predictor. Compiler and source-to-source transformation techniques [28, 73] have also been developed to thwart side-channel attacks.

Wang et al. [111] identified the causes of the timing attacks to be the underlying hardware. In particular, their work focuses on specialized cache designs, such as Partition-Locked Caches [112] and Random Permutation caches [111] that defend against caching attacks in hardware. Very recently, Coppens [28] discovered two previously unknown attacks on the complex pipeline run time variance of x86 architectures.

Our work builds upon the experiences of these. Most solutions employ either exclusively hardware or software techniques to defend against attacks. We recognize that a complete solution to control temporal semantics requires a combination of both software and hardware approaches to defend against and prevent future side-channel attacks. Hence, we present an effort that includes timing control instructions to control execution times in software, and a predictable processor architecture to realize the instructions. By doing this, we completely eliminate the source of leaked information used by time-exploiting attacks, rendering the system immune against such attacks.

### 4.2.2 A Precision Timed Architecture for Embedded Security

The foundation of time-exploiting attacks exploits the uncontrollable timing variability introduced to programs by underlying the implementation of encryption algorithms. Software implementations naturally introduce varying run times because of data-dependent control flow paths. Modern computer architectures create unpredictable execution times by abstracting away hardware optimizations meant to improve average case performance. In this section we will present several features of PRET that bring *controllability* over timing to software, eliminating the origin of the attacks. We will discuss the software extensions that allow timing specification in programs, and the predictable architecture to comply with these specifications. These two approaches cannot be separated. A predictable architecture by itself would only ease the feasibility of an attack, and software timing specifications are meaningless if they cannot be met by the hardware. By combining both hardware and software solutions, we yield a timing predictable and controllable architecture. Thus, by design, PRET prevents leakage of any timing side-channel information, and eliminates the

core vulnerability of time-exploiting attacks.

**Controlling Execution Time in Software**

It is extremely difficult to control and reason about timing behaviors in software, even with adequate understanding of the underlying architecture. Current instruction-set architectures (ISA) have neglected to bring the temporal semantics of the underlying architecture up to the software level. Thus, architecture designs have introduced clever techniques to improve on average case execution time of the instructions, at the expense of introducing variability in instruction execution time. These architecture improvements are hidden to the software behind the abstraction of the ISA. This proves to be costly in terms of security, because it uncontrollably leaks timing information which can correlate to the secret key.

In section 2.3 we introduced several ISA extensions that add time controlling behaviors to software. The extensions provide timing instructions that enable a programmer to have more control of execution time in software. These instructions do not physically alter processor speed, or modify the execution time of instructions on the architecture. Instead, they are meant to aid the programmer in dealing with timing variability from data-dependent control flow paths by allowing the programmer to interact with various execution time behaviors in software. This includes the ability to specify a desired execution time for code segments, and the ability to detect and handle situations when the execution time exceeds the desired amount. Specifically in this context, the ability to enforce a minimum execution time for code segments proves extremely useful for mitigating the varying execution speeds exhibited by algorithms or code segments. We showed in section 4.1 how the *delay_and_set* instruction can be used to synchronize execution and communication of different nodes for an implementation of a real-time 1D-CFD simulation. Encryption algorithms can exhibit varying execution time behaviors depending on the bits of the encryption key. The algorithm follows different execution paths if a particular bit in the key is set or not, allowing attackers to exploit this execution time variance to obtain the key. By using the timing instructions provided by the PRET architecture, we can mitigate the effects of this, eliminating the exploit causing this timing attack.

At the expense of more programming effort, other solutions have been proposed to alter and pad the execution time of different execution paths [52] to shield against the timing variability of the algorithm. At a glance it might seem that the timing instruction are a similar solution to these proposals, however, the principles are inherently different. While effective against certain time-exploiting attacks, existing solutions alter the underlying algorithm implementation in attempt to manually pad or distort the execution time. These solutions are not only algorithmically specific, but could lead to unnecessarily degrading of the performance of encryption algorithms. The timing instructions, on the other hand, allows for a separation of concern between the functionality and timing behavior of the code. The programmer can implement the correct functionality of the algorithm, then use timing instructions to regulate its timing behavior. The subtle difference will be more apparent in section 4.2.3 when we show two different implementations of the RSA encryption that both use timing instructions to regulate execution time. One implementation mimics existing execution time padding solutions, and the second implementation uses timing instructions to enforce an overall execution time of the RSA algorithm. We present performance comparisons and show that explicit timing control instructions could prove more beneficial than simple execution time padding.

The timing instructions provide a method to control the timing behavior of a program in software. However, they do not change the behavior of the underlying architecture. If the underlying architecture makes the reasoning of execution time difficult, then these instructions become more difficult to use. Timing instructions alone do not prevent attacks that exploit architectural designs to inject execution time variances [81, 4] and obtain side-channel information. We argue that a *predictable* architecture is also required to eliminate timing exploiting attacks.

### Predictable Architecture

**Pipeline** In order to improve instruction throughput and performance, modern processor architectures implement pipelines to execute multiple instructions in parallel. This requires handling of pipeline hazards, which are caused by dependencies in instruction sequences. Conditional branches are the perfect example – the pipeline cannot fetch and begin executing the next instruction without knowing which instruction to fetch. Since a conditional branch usually takes more than one cycle to resolve, the processor is forced to stall until the branch is resolved.

Computer architects use clever speculative techniques to mitigate the effects of pipeline hazards and to substantially improve the average-case performance. For example, branch predictors are used to guess the next instruction needed by the processor for branches [36]. This allows the processor to execute instructions speculatively while rolling back only when needed. While these speculative techniques improve the average-case performance, they introduce several side effects. First, they create *timing variations*. Depending on the outcome of its speculation, the processor might need to discard the wrongly speculated work, and re-execute the correct instructions. Second, these units are *unpredictable*. Since these units are shared by all software processes concurrently running on the processor, the states of speculation units are heavily dependent on the different interleaving of processes. This means that a process can unknowingly be affected by other processes, since the speculation state is shared between them [57]. Because the goal of these speculation techniques is to improve program performance without effort from the programmer, the controls of these speculation units are concealed from the programmer, and cannot be directly accessed in software. Thus, these side effects result in *uncontrollable* timing behaviors in the program.

Several Multithreaded architectures enable more opportunities to exploit the uncontrollable timing behaviors. Multithreading utilizes thread-level parallelism by introducing multiple hardware threads in the processor. This allows the execution of another hardware thread during pipeline stalls like branches or memory accesses. However, typical multithreaded architectures share hardware units effecting execution time between hardware threads, enabling threads to covertly affect other threads execution time. The class of Simultaneous Multithreading (SMT) architectures presents an example of this. Here, the hardware threads share multiple execution units and execute in parallel depending on a hardware scheduler. Attackers exploit such designs by running a spy thread that executes concurrently with a thread that implements the encryption algorithm. This spy thread probes the components shared with the encryption thread [81, 4] by forcefully occupying the shared units and observing when they are evicted by the encryption thread. The announcement of this vulnerability caused Hyper-Threading, Intel's implementation of SMT, to be disabled by default in some Linux distributions because of its security risks [82]. For general purpose applications, these side effects pose insignificant threats, but for security applications, the consequences are uncontrollable sources of side-channel information leakages.

The PREcision Timed (PRET) architecture is a timing predictable architecture proposed

for real-time embedded systems. As discussed in chapter 2.1.3, PRET employs a thread-interleaved pipeline, a multithreaded pipeline that employs a predictable round-robin thread scheduling policy between the hardware threads every cycle. Instructions from each thread are predictably fetched into the pipeline every $n$ cycles, where $n$ is the number of hardware threads. If $n$ is greater than the number of stall cycles needed for data dependency hazards, then we effectively remove those hazards because the data value is available during the next cycle in which the thread is dispatched. For example, if we set $n$ to be the number of stages in the pipeline, then we eliminate the need for any data forwarding/bypassing logic, along with the need for hardware speculation units such as branch predictors. Most importantly, the hardware threads are temporally isolated, meaning that no threads can affect each others timing behavior. Each individual hardware thread maintains their own copy of the processor state (program counter, general purpose registers, stack pointer, etc.), and each hardware thread runs independently with no shared state in the pipeline. Because of the simple and transparent thread-scheduling policy, each hardware thread gets dispatched in a predictable way that cannot be affected by other hardware threads. Thread-interleaved pipelines allow us to gain higher instruction throughput without the harmful side effects.

**Memory System** The memory system presents another opportunity for attackers to gain side-channel information. The high clock speed of modern processors combined with the high latency to access main memory results in sometimes hundreds of cycles stalled when the processor needs to access the main memory. On-chip fast access memories are used to bridge this access latency, creating a *memory hierarchy*. Caches are *hardware-controlled* fast-access memories that predict and prefetch data from main memory based on temporal and spatial locality of data accesses from the processor. If the cache control speculation is accurate, then access to data can complete in one cycle, and no stall in the pipeline is required. However, when a misprediction occurs, data needs to be fetched from the main memory, causing a drastic difference in the access time [104]. Caches abstract away this memory hierarchy and access latency variation from the programmer by managing the cache contents in the hardware. Because threads and processes share the same memory system, attackers can probe the memory access patterns of the encryption process by evicting shared cache lines and observing the timing variation it causes [81]. This is possible because the memory hierarchy is abstracted away from the programmer, resulting in *uncontrollable* timing behaviors.

PRET utilizes scratchpads memories (SPM) instead of caches in its memory hierarchy. SPMs are fast access memories controlled by software. SPMs use less power and occupy less area [14] because no speculation logic is needed. SPMs occupy a distinct address space, which exposes the memory hierarchy to the programmer, instead of abstracting it away like caches. The allocation of data between memory and SPM is done with explicit instructions, either at compile time by the compiler or manually by the programmer. This gives the software control over memory access latencies, and provides a predictable execution time of the program. The performance of SPMs vary based on the data access patterns of the application, but since the control is in software, it is possible to tune the allocation scheme to achieve even better performance than a generic cache for specific applications. There are abundant ongoing research on allocation schemes and methods for optimizing the performance of SPMs [13, 15, 79, 101, 106]. SPMs can be found in the Cell processor [37], which is used in Sony PlayStation 3 consoles, and NVIDIA's 8800 GPU, which provide 16KB of SPM per thread-bundle [76]. Although this comes at the cost of more programing effort, but it is not uncommon to see platform specific tuning of software for performance purposes.

For example, high performance parallel algorithms are often fine tuned to work on block sizes depending on the cache size and replacement policy of the platform, and re-tuned when running on different platform.

For security purposes, the scratchpad on PRET is configured to provide each hardware-thread a private scratchpad region so the scratchpad contents cannot be modified or monitored by spy threads on running another hardware thread. This prevents shared resource time-exploiting attacks on the fast access memory across hardware threads. Even if an encryption process is sharing a hardware thread with another process, the contents of the scratchpad is controlled in software or statically compiled in by the compiler. The thread managing supervisor code can manage the contents on the scratchpad before the processes are scheduled and unscheduled, preventing a spy process from affecting the execution time of the encryption process. Clearly, the edge that SPMs give over conventional caches is their *controllability* in software, thus preventing unwanted timing side-effects from attackers and spy threads, even though the SPM is shared by software processes.

Although no known attacks have exploited main memory access, typical DRAM controllers also result in variable memory access latencies, and are shared amongst all threads and processes within the system. A predictable DRAM controller is designed and interfaced with the thread-interleaved pipeline of PRET to provide predictable memory access latencies to all threads. The DRAM controller privatizes DRAM bank resources to remove bank conflicts and fully utilize bank level parallelism on the DRAM. Each hardware thread in the thread-interleaved pipeline is mapped to a privatized DRAM bank resource. On the backend, the bank resources are accessed in a round robin order fashion, to remove temporal interference between accesses to the bank resources. All memory accesses from the hardware threads are isolated from each other, removing any possibilities of cross-thread side-channel attacks from the shared memory controller. The DRAM memory access latencies are decoupled from the data access patterns, thus, even processes on the same hardware thread that access the same bank resources cannot alter each others execution time in attempt to gain side-channel information. More details on the PRET DRAM controller is presented in section 2.2.2.

We acknowledge the many efforts to counteract timing attacks with algorithm rewrites to control and balance the run time of the algorithm. These efforts while successful, are ad-hoc, counteracting specific attacks without prevention of others. Without tackling the origin of time-exploiting attacks, we believe that more exploits will eventually be discovered, attacking the *uncontrollable* execution time variation caused by the shared resources of hardware or software control flow. The PRET architecture is designed to ensure repeatable and predictable timing behavior of programs by providing control of timing properties in software and a predictable architecture that provides temporal isolation for hardware threads and processes. PRET is impenetrable known attacks such as branch predictor attacks [5], cache attacks [81] or other attacks on the pipeline [28]. The more importantly, the predictable architecture design removes the root cause of time-exploiting attacks – the *uncontrollable* timing variations caused by unpredictable hardware components or software control flows.

### 4.2.3 Case Studies

In the following section we will show results of two encryption algorithms running on PRET. All experiments are run on the cycle accurate simulator of the PRET architecture described in [62]. The simulator implements the SPARC v8 instruction set, and employs six threads on a

six stage thread-interleaved pipeline. Programs are written in C and compiled using a standard gcc cross compiler from Gaisler research labs [34]. This PRET implementation implements a simple processor extension inspired by Ip and Edwards [45] that adds timing instructions to the ISA. To be consistent with the terminology used in [45], we call this instruction the *deadline instruction*. This deadline instruction has similar semantics to the *delay_and_set* instruction introduced in section 2.3. It first ensures the previous deadline specified is met, then sets the deadline for the next instruction sequence. The deadline instruction specifies time in units of thread cycle, which is a thread's perceived cycle.

### RSA Vulnerability

The central computation of the RSA algorithm is based primarily on modular exponentiation. This is shown in algorithm 1. Of the inputs, $M$ is the message, $N$ is a publicly known modulus, and $d$ is the secret key. Depending on the value of each bit of $d$ on line 4, the operation on line 5 is either executed or not. This creates variation in the algorithm's execution time that is dependent on the key, as mentioned in [52].

```
   Input: M, N, d = (d_{n-1} d_{n-2} ... d_1 d_0)
   Output: S = M^d mod N
1  S ← 1
2  for j = n - 1 ... 0 do
3  |   S ← S^2 mod N
4  |   if d_j = 1 then
5  |   |   S ← S · M mod N
6  |   return S
```

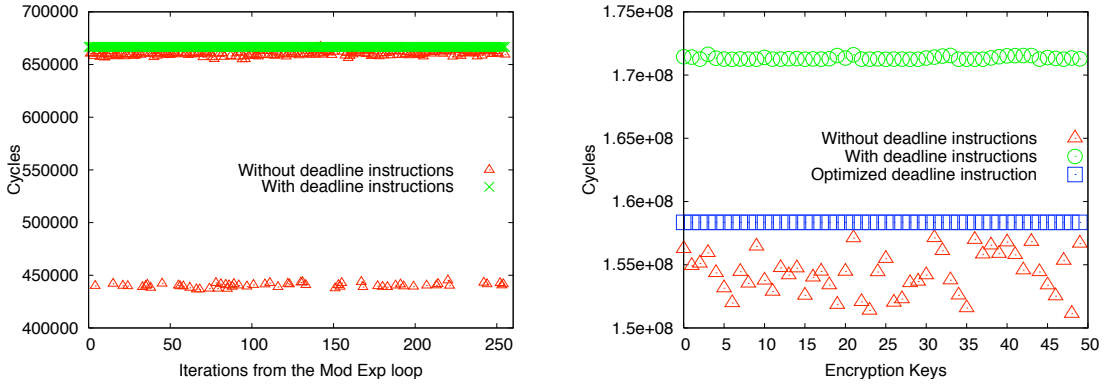
**Algorithm 1:** RSA Cipher

```
   Input: M, N, d = (d_{n-1} d_{n-2} ... d_1 d_0)
   Output: S = M^d mod N
1  S ← 1
2  for j = n - 1 ... 0 do
3  |   /* 110000 is 660000÷6 cycles, since deadline registers
   |   are decremented every 6 cycles.*/
4  |   dead(110000);
5  |   S ← S^2 mod N
6  |   if d_j = 1 then
7  |   |   S ← S · M mod N
8  |   dead(0);
9  |   return S
```

**Algorithm 2:** RSA Cipher with deadline instructions

When the reference implementation of RSA (RSAREF 2.0) was ported to the PRET architecture, single iterations of the loop varied in execution time almost exclusively due to the value of $d_j$, which is the $j^{th}$ bit of the key. The triangle points in figure 4.8(a) show the measured run time of each iteration in the for loop (lines 2–6) in algorithm 1. Each iteration took approximately either 440 or 660 kilocycles, with very little deviation from the two means. As a simple illustration, we can fix the execution time of each iteration in software by adding deadline instructions in the body of the loop as shown in algorithm 2. When enclosed with deadline instructions, the execution time of each iteration is uniform, and the bimodality of the execution time is completely eliminated. The x points in figure 4.8(a) show the measured time of each iteration after adding deadline instructions; they are simply a straight line.

We observe the large-scale effect of this small change on the whole encryption in figure 4.8(b), where RSA was run fifty times using randomly generated keys. Without the deadline instructions (triangle points), different keys exhibit significant diversity in algorithm execution time. With the deadline instructions added within the modular exponentiation loop (circle points), the fluctuation is dramatically reduced to almost none. The remaining small variations result from code that is outside of the modular exponentiation loop, which is not influenced by the actual key. From figure 4.8(b) we can see that this small variation is not significant enough to correlate the total execution time and the key.




(a) Run time of Modular Exponent operation

(b) Run time of RSA operation

Figure 4.8: RSA Algorithm

Without explicit control over timing, any attempt to make an algorithm run at constant time in software would involve manual padding of conditional branches. This forces the algorithm to run at the worst-case execution time, similar to what we've showed. As a result, although this makes the encryption algorithm completely secure against time-exploiting attacks, they are not adopted in practice because of this overhead. Nevertheless, with control over execution time, we will show that running encryption algorithms in constant time does not necessarily require it to run at the absolute worst-case execution time.

### An Improved Technique of using Deadline Instructions

It is expected that the distribution of RSA run times will be normal over the set of all possible keys [52]. Figure 4.9 shows the run time distribution measured for one thousand randomly generated keys. A curve fitting yields a bell shaped curve formed from the run time distribution of all keys. This means that the execution time of approximately 95% of the keys will be within $\pm 2$ standard deviations of the mean, and the worst-case execution time will be an outlier on the far right of this curve. Our previous example fixed the execution time of all keys to be *roughly* at this far right outlier. An improved technique capitalizes on this distribution of run times to improve performance.

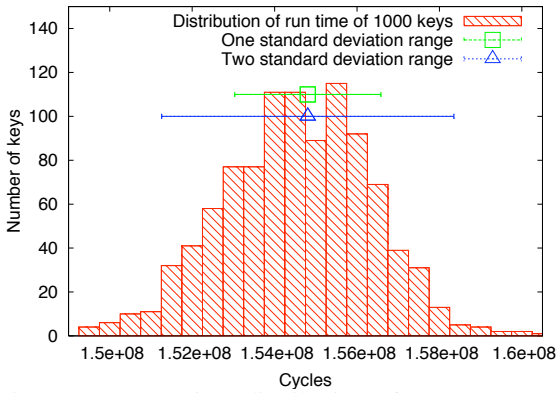


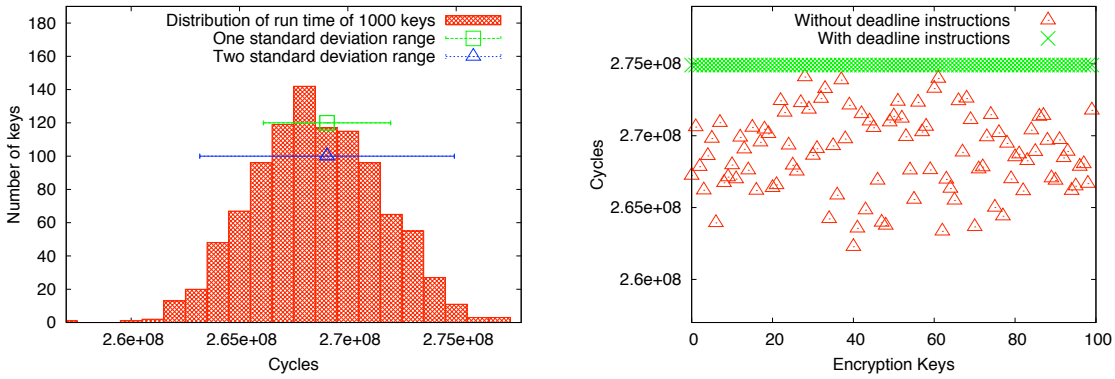

Figure 4.9: Run time distribution of 1000 randomly generated keys for RSA

First, instead of enclosing the loop iterations of the modular exponentiation operation, we enclose the whole RSA operation with deadline instructions. Now the deadline instructions are used to control the overall execution time of the RSA operation. Note that we could have done this for the previous example as well to fix the execution time to be *exactly* the worst-case, always.

For RSA, key lengths typically need to be longer than 512 bits to be considered crypto-

graphically strong [87]. This gives roughly $2^{512}$ possible keys, which is far more than needed for most applications. Suppose we are able reduce the key space the application covers – instead of using $100\%$ of the keys, we refine our encryption system to only assign $97\%$ of all possible keys. Namely, the subset of keys whose RSA execution times fall on the left of the $+2$ standard deviation line on the curve. Statistically, the keys that lie outside of $\pm 2$ standard deviation are the least secure keys anyway, since it is easier for time-exploiting attacks to distinguish those keys. By doing so, we reduce the execution time of the encryption algorithm because we know that keys that are right-side outliers will not be used.

With timing control in software, we can take advantage of this information by simply reducing the value specified in the deadline instructions enclosing the whole RSA operation. The square points in figure 4.8(b) show the results of using deadline instructions in this way. We reran the same fifty keys from the previous section, and enclosed the whole operation with deadline instructions that specified the run time at +2 standard deviations from the bell curve we obtained. We can see that, compared to the previous results that fixed the execution time of each key to take the worst-case time (circle points), we clearly reduced the overhead while still running in constant time. By taking the run time difference between executions with and without deadline instructions, we obtained the overhead introduced for each of the keys with run time below 2 standard deviations (97.9% of keys in our case) within the one thousand key set in our experiment. This calculation reveals that by merely reducing the key space by 3%, running the encryption with optimized deadline instructions only introduced an average overhead of $2.3\%$ over all the keys we measured. All this while still being completely immune to time-exploiting attacks. This is virtually impossible to achieve without explicit timing control, which illustrates the value of decoupling timing control and functional properties of software.




(a) Distribution of 1000 DSA keys

(b) Run time of 100 DSA operations

Figure 4.10: Digital Signature Standard Algorithm

### Digital Signature Algorithm

Kocher's [52] original paper mentioned that Digital Signature Standard [75] is also susceptible to timing attacks. Thus, to further illustrate our case, we ported the Digital Signature Algorithm from the current OpenSSL library (0.9.8j) onto PRET. We used the same method mentioned above to secure this implementation on PRET. Figure 4.10(a) shows the distribution of DSA

run time for one thousand keys. It also shows a normal distribution. Then, we randomly generated another one hundred keys, and measured the run time with and without deadline instructions, which we show in figure 4.10(b). We can see clearly that the run time with deadline instructions is constant, and any time-exploiting attack is not possible.

Currently, we do not know of any work that correlates the key value with run time for different encryption algorithms. However, with the ability to control execution time in software, such a study would be extremely valuable. Figures 4.9 and 4.10(a) show that RSA and DSA follow a normal distribution. Thus, from the algorithm, we postulate that by simply counting the $1$ bits in the key should be sufficient to distinguish the $95\%$ of secure keys before assigning. Note that no change to the encryption algorithm itself is needed, but only the key assignment process. Since we can adjust the execution time in software, we can tune the performance of each application based on the application size, key bit length and performance needs. All this can be done while maintaining complete immunity against time-exploiting attacks.

Note that there are several other software techniques specific to encryption algorithms that successfully defend against timing attacks. Our work does not lessen or replace the significance of those findings. Instead, we can use traditional noise injection defenses on PRET as well. For example, if reducing the key space is not possible for some applications running RSA then RSA with blinding can be ran on PRET. By simply running on PRET, the encryption algorithm is also secure against shared hardware resource attacks such as caches, and branch predictors. Other encryption algorithms that do not have software techniques or solutions readily available to counteract timing attacks can easily use the deadline instructions provided by PRET to achieve security against timing attacks.

### 4.2.4 Conclusion and Future Work

Side-channel attacks are a credible threat to many cryptosystems. They exist not just because of a weakness in an algorithm's mathematical underpinnings, but also from information leaks in the implementation of the algorithm. In particular, this paper targets time-exploiting attacks, and lays out a means of addressing what we consider the root cause of such attacks: the lack of *controllability* over the timing information leaks. As an architecture founded on predictable timing behaviors, PRET provides timing instructions to allow timing specifications in software. In addition, PRET is a predictable architecture that guarantees removes timing interference through a thread-interleaved pipeline with scratchpad memories for each hardware thread, and a predictable DRAM memory controller. This eliminates the shared states in the architecture that create uncontrollable timing interference, exploited by attackers. Through a combination of hardware and software techniques, PRET gives control over the timing properties of programs, which effectively eliminates time-exploiting attacks.

We demonstrate the application of these principles to known-vulnerable implementations of RSA and DSA, and show that PRET successfully defends against time-exploiting attacks with low overhead. Our work does not undermine the significance of any related work, which have mostly been specific to certain attacks. PRET does not target a specific encryption algorithm, because it can be used in combination with these partial solutions on specific encryption algorithms, as well as provide a complete defense for other encryption algorithms which are less researched upon.

Besides time-exploiting attacks, there are other side-channel attacks that are legitimate

threats to encryption algorithms such as power, and fault attacks. We plan to continue to investigate PRET's effectiveness in defending against them. We conjecture that the thread-interleaved pipeline used in PRET can potentially help defend against power attacks because the power measured from the processor now includes significant interference from the execution of other hardware threads in the architecture.

# Chapter 5

# Related Work

We are certainly not the first or only one to tackle the unpredictability of computer architecture designs. In this chapter we survey the abundance of related research to our goal of predictable architectures. Timing analysis techniques, compiler techniques and architectural techniques all play a role in tackling the unpredictability of computer architectures. However, we limit the scope of this survey to a few compiler techniques with the main focus on architectural techniques, as that is the focus of this thesis.

Adding temporal semantics to programming languages has been the focus of many research proposals, but to the best of our knowledge, we believe this is the first attempt to introduce temporal semantics down to the ISA level.

## 5.1 Pipeline-Focused Techniques

### 5.1.1 Static Branch Predictors

Dynamic branch predictors cause timing anomalies [32], and are difficult to model is because of the *aliasing* of branch points. *Aliasing* occurs when two different branches occupy the same branch predictor slot, and cause interference. Burguiere et al. [24] made a case for *static branch prediction* to be used for real-time systems. This could be done in several ways. The simplest form can predict all branches taken or not taken. Improvements can include the *Backward Taken, Forward Not taken* scheme, to improve performance for loops and if statements. This uses the observation that for loop control branches, most all backwards branches are taken to return to the loop, and only at the end of the loop are forward branches taken. With architecture support for static branch predictions, compilers can analyze code patterns (loops, if-then-else, if then) and insert instruction set constructs to denote the static prediction of each branch. The underlying architecture will use that for its prediction, instead of relying on a dynamic hardware unit. This removes *aliasing* and gives better estimated worst case branch mispredicts.

Bodin et al. [21] uses this idea of software static branch prediction to improve WCET of programs. Intuitively, they aim to remove all branch mispredict penalties from the worst-case path to improve the WCET. They propose an algorithm that iterates through the control flow graph to find the worst-case execution path (WCEP). Initially, the algorithm finds the worst case path assuming all the branches are mispredicted. Then, the algorithm assigns the static branch prediction

of all branches on the WCEP to be taken. The algorithm then iterates again to find the new WCEP until two iterations yield the same WCEP. Since the algorithm never reassigns assigned branches, it always converges but is not optimal. The presence of caches can easily effect the WCEP, and each branch prediction reassigned could modify the cache state. However, the experiments assumed all code and data fit into the caches, thus the effect of caches were not factored into by the algorithm.

### 5.1.2 Superscalar Pipelines

Superscalar pipelines issue multiple instructions at a time to exploit instruction-level parallelism (ILP). In order to keep the pipeline filled, superscalar pipelines typically employ more aggressive techniques to fully utilize the ILP. As a result, attempting to model all advanced techniques often leads to either very pessimistic results, or almost infeasible complex models.

Rochange et al. [91] propose to use instruction pre-scheduling to ease the difficulties of analysis of superscalar pipelines. The concept is similar to resetting the pipeline state before each basic block execution. This is done by postponing the scheduling of instructions from the next basic block until the instructions from the previous basic block are completed. If it is possible to remove all timing interference across basic blocks, then the resources needed to model the pipeline could be significantly reduced, as each basic block will start with a consistent initial state. However, the results assume the absence of caches, which could easily effect execution across basic blocks. Furthermore, depending on how many instructions can be in flight at one time, waiting for the pipeline state to be flushed could induce large penalties for programs with a lot of control flow transfer and small basic blocks.

Whitham et al. [116] combined the techniques of *instruction pre-scheduling* and *static branch predictions* and propose modifications to an out-of-order superscalar pipeline to provide predictability for single thread execution. Instead of basic blocks, the superscalar pipeline pre-schedules instructions across *virtual traces*[115]. *Virtual traces* are program paths with static branch predictions inserted. These are usually formed by predicting along the WCEP, similar to the algorithm Bodin et al. [21] introduced. Each virtual trace can contain a fixed number of branches. A VTC (virtual trace controller) is introduced to control the progress of the pipeline. The VTC contains a VTR (virtual trace register) which stores the branch predictions, and the pipeline state is reset between traces so the WCET analysis can be limited to within traces. The out-of-order superscalar pipeline is also modified to disallow memory prediction, reordering of branches, and assume scratchpads are used instead of caches. This allows the execution of traces to run predictably for each different exit (branch mispredict) within a trace. They show an improved throughput for most programs when compared to a simple in-order CPU model, and studied the effects of trace sizes in order to balance the main path execution time against the costs of side exits

### 5.1.3 VLIW architectures

VLIW machines, like superscalars, issue multiple instructions at a time to exploit ILP. However, unlike superscalrs, they rely on compiler to utilize ILP and determine the instructions issued. This helps in the predictability of the software because the hardware does minimum reordering or stalling.

Yan et al. [124] studied the predictability of VLIW machines, and propose changes to the architecture and compiler to improve the predictability. They found that although most of the

data dependency is scheduled away by the compiler, there are still several factors that limit the predictability on the hardware. First, memory access latency will still cause instructions to stall from the architecture. Since statically it is not known whether a memory access is a hit or a miss, the hardware still needs to check and stall for it. Second, data dependency still exists across compilation units, so the hardware still supports basic data dependency checking to handle those dependencies. A compilation unit could be a basic block, a loop, a procedure or a region [38]. Finally, if the VLIW uses branch prediction, there is still the need for handling of mis-prediction etc.

As VLIW machines heavily utilize the compiler to improve performance, they propose several compiler techniques to compile programs that lend themselves to better WCET for VLIW architectures. First, they use the single-path paradigm proposed by Puschner and Burns [85], and eliminate all non-loop backwards branches with full if-conversions [10]. To mitigate the performance penalty of single-path programming with aggressive hyperblock scheduling [65] to exploit the ILP from VLIW architectures. For the data dependencies across compilation units, they use code padding to ensure the execution time is consistent across different paths. This will enable easier WCET analysis. This work minimally deals with instruction caches, but does not account in the effects of data cache.

### 5.1.4 Multithreaded Pipelines

#### Thread Scheduling

With explicit hardware multithreading, the scheduling policy plays a huge role in the predictability of the architecture. Kreuzinger et al. [55] evaluated the use of different real time scheduling schemes to schedule hardware threads to handle external events. They evaluated fixed priority preemptive (FPP), earliest deadline first (EDF), least laxity first (LLF) and guaranteed percentage (GP), which is essentially time sharing the pipeline. The architecture used for evaluation is a java multi-threaded superscalar pipeline with four threads [54]. A hardware priority manager is implemented to facilitate the scheduling of threads. All real-time threads registers its real-time requirements during initialization stage to the priority manager. When the external event occurs, the priority manager schedules the corresponding interrupt service thread, and starts assigning priorities based upon the real time requirements. The evaluation criteria to compare scheduling policies is the throughput of the processor. The conclusion of the report is that in order to maximize multiple threads on a superscalar machine, the scheduler should try and keep as many threads active as long as possible to leverage thread level parallelism and hide more latencies of pipeline stalls. Thus GP does the best because it schedules different active threads each cycle until their percentage runs out, thus it keeps threads alive as long as possible. The idea of using hardware threads to service interrupts is novel because of the low overhead to switch contexts. By giving the interrupt service routine thread priorities, it may be possible the bound the execution time of higher priority threads. Although the dynamic thread scheduling can cause execution time bounds to be imprecise from the effects of timing interference across threads.

El-Haj-Mahmoud et al. [31] proposed a statically scheduled multithreaded architecture called the Real-Time Virtual Multiprocessor(RVMP). The idea of a virtual processor is a slice of time on the process. The RVMP extends an in-order 4-way superscalar processor to support the partitioning of the pipeline in *space* and *time*. In the *space* dimension, the resources of the superscalar can be partitioned to different threads. In the *time* dimension, the superscalar resources

are time shared, and different threads are scheduled to utilize the resources at different times. The hardware extensions to the superscalar pipeline prevent interference between the virtual partitionings. Scratchpads are employed for predictable memory access latencies, although they assume all accesses go to the scratchpad. It is unclear how accesses to shared resources, in particular main memory are dealt with. A static round-based schedule of the thread execution is constructed to account for the real-time requirements of each thread. The static schedule utilizes the flexibility of the different time and space partitioning options to allow threads with higher utilization more access to the pipeline.

**Simultaneous Multithreaded Architectures**

Simultaneous Multithreaded Architectures (SMT) attempt to exploit both instruction-level and thread-level parallelism by dynamically scheduling multiple hardware threads onto a multi-way pipeline. Each cycle, instructions from different threads can be fetched simultaneously to fully utilize the pipeline. The dynamic scheduling and aggressive speculation techniques render SMTs almost impossible to use for real-time systems. However, several proposals involved slight modifications to architecture to create a *WCET-aware* SMT to be used for real-time systems.

Barre et al. [16] proposed to assign one explicit hardware thread with the highest priority. That thread, called the real-time thread, gains access to any resource whenever it is scheduled. Any other thread that is currently occupying the resource will be preempted, and later replayed when the real-time thread is not using it. The modifications to the SMT include additions to allow the preemption, and also the partitioning of any resource that needs to be shared. This gives the highest priority thread the illusion that it has the whole superscalar pipeline to itself, reducing the execution time analysis of the real-time thread to the equivalent of a superscalar architecture. Currently the cache effects and branch prediction are listed as future work.

Hily et al. [44] showed that out-of-order execution may not be as cost effective as in-order execution on SMT machines. Thus, Uhrig et al. [93] proposed a similar concept to Barre et al. [16], except for an in-order executed superscalar. Mische et al.[71] expands this to allow more than one real-time thread running on the SMT architecture. They do so by essentially time-sharing the highest priority thread slot amongst the real-time threads. The time-sharing schedule is statically constructed to ensure that the real-time threads still provide reasonable WCET guarantees. This architecture uses instruction scratchpads without data scratchpads, and no branch predictors, as the branch penalty can be filled with executions from other threads. Some issues do arise with the contention of memory access, as it is difficult to partition memory accesses between hardware threads. Contention between the high priority thread slot and other thread slots are resolved by alerting the memory controller from earlier stages in the pipeline that a high priority thread will issue a memory instruction. This way the memory controller can hold off service to the lower priority memory accesses and wait for the high priority access to come. Contention between the real-time threads on the high priority slot however are not resolved.

### 5.1.5 Others

#### Virtual Simple Architecture

Anantaraman et al. [11] propose the virtual simple architectures (VISA), which uses dynamic checking to ensure tasks are meeting the deadlines. The microarchitecture is split into two modes. A simple mode, which conforms to the the timing of a hypothetical simple pipeline that is amenable to safe and tight WCET analysis. And a high performance mode, which the architecture can use arbitrary performance-enhancing features. A task that executes on the VISA is divided into multiple sub-tasks, to gauge progress on the complex pipeline. Each sub-task is assigned an interim deadline, based on the hypothetical simple pipeline. When tasks are executed on the VISA, they are first speculatively executed in high-performance mode. If no checkpoints are missed, then the high performance mode has met the timing requirements. If a checkpoint is missed, the architectures switches to a simple mode to bound the remaining task times to attempt at meeting the timing constraints. The results showed that the high performance mode had average executions times of 3 to 4 times faster than the simple mode, and dicussed possible power savings by scaling the voltage in high performance mode. However, the tasks and programs must have sufficient slack time to allow for dynamic checking of deadlines, and it is unclear if the simple mode will always be able to make up the time if the high performance mode missed its checkpoint.

#### Java Optimized Processor

Schoeberl presented the Java Optimized Processor (JOP) [97] which uses Java for real time embedded systems. The design of JOP includes a two level stack cache architecture [96]. Instead of using a large register file to store the stack like in PicoJava[67], it only uses two registers to store the top two entries of the stack (Register A and Regiser B). Leveraging the stack based architecture of JavaVM, whenever an arithmetic operation occurs, the result is always stored back to the top of the stack (Register A). Any push or pop operation simply results in a shift of values between the two registers and the stack cache, which only requires one read and one write port for the memory. This architecture does not have any data hazards and has very few pipeline stages (no need for an explicit commit/writeback stage). Because of the few pipeline stages, it only has a small branch delay penalty, so no branch predictor is used. All bytecode on JOP is translated into a fixed length microcode. Each microcode executes in a fixed amount of cycles, independent of its surrounding instruction. Thus, the WCET analysis only requires a lookup table of bytecode translated into microcode, rendering it a predictable architecture.

#### MCGREP

Whitham introduced the Microprogrammed Coarse Grained Reconfigurable Processor (MCGREP) [114], which is a reconfigurable predictable architecture. The architecture of MCGREP contains multiple execution units, but each operation is implemented in microcode. The pipeline architecture is extremely simple, reassembling a two stage pipeline with a fetch/decode stage and an execute stage. No internal state is stored in the pipeline, and instructions do not affect each others execution time. A fast internal RAM without cache is used to store the program and be used as memory for data. The microcode operations are predictable in the MCGREP architecture, taking a fixed number of cycles to complete. Advanced operations can be dynamically loaded as new

microcode, which enables application specific instructions to improve performance. All MCGREP instructions take a fixed number of clock cycles to complete and are unaffected by execution history, making MCGREP a predictable processor.

## 5.2 Memory-Focused Techniques

### 5.2.1 Caches

The dynamic behavior of caches cause headaches for real-time systems when trying to predict memory access latencies. Reineke et al. [88] presented a study on the predictability of different cache replacement policies. They evaluate the Least Recently Used (LRU), First In First Out (FIFO), Pseudo LRU (PLRU) and Most Recently Used (MRU) replacement policies to determine if LRU was more predictable than other policies, as observed by Heckmann et al. [40]. The results confirmed that the LRU replacement policy was significantly more predictable than other policies, and recommended any real-time system with caches to use LRU for its replacement policy. It also revealed potential for improvement in existing analyses of PLRU and FIFO.

Puat and Decotigny [83] proposed to use partitioned and lock caches to eliminate the intra- and inter-task interferences when a cache is used. Intra-task interferences occur when different memory blocks of the same task compete for cache blocks. Inter-task interferences occur where a preempting task's memory blocks cause cache reloads in the preempted task. By using cache partitioning, a part of the cache is reserved for a particular task, and inter-task interference is eliminated. To eliminate intra-task interference, cache locking is used to lock the contents of cache. The cache contents can be locked statically, which are fixed at system start for the whole run time, or dynamically, where the contents may change. By locking and partitioning caches, the memory access latencies will have more predictable behavior.

Schoeberl [95] proposed to use method caches for the instruction cache of the JOP architecture [97]. Conventional caches use a cache line as its basic unit of replacement. Method caches use *methods* as its unit of replacement. A cache can contain different block sizes that are used to store methods. There exists a trade off between performance and predictability for the block sizes of the method cache. Methods can occupy more than one block, depending on the method size. When a method is called, the cache loads the whole method into the cache, occupying any number of blocks it needs. The LRU replacement policy is used, since the end of a method usually returns to its parent method. When a method is evicted, all blocks it occupies are evicted. Thus, the instruction cache is more predictable, because it only changes on method calls. Within a method, all instructions are known to be in the cache, so no cache miss results from the instruction cache.

Metzlaff et al. [69] used a method cache mechanism with the real-time SMT architecture in [71]. They partitioned the scratchpad for each different thread so no inter-thread interference would exist. Then, they implemented the method cache [50] with scratchpads, and give priority to the high priority thread when a filling is needed. THey called this the function scratchpad. If the thread is stalled when a method is being filled into the scratchpad, other threads occupy the pipeline, so throughput is preserved with multiple threads.

### 5.2.2 Scratchpads

Scratchpads are known to allow more precise WCET analysis [113] because the contents are managed in software. Puaut et al. [84] presented a comparison of locked caches and scratchpads, and showed that were only subtle differences between the two in terms of performance. Most benchmarks provided similar WCET estimates. The difference stems from the granularity of allocation units. For locked caches, the basic allocation unit is a cache line. Thus, it is possible to *pollute* the contents of the cache line with contents that are not part of the allocation scheme. Also, depending on the associativity of the cache, a cache line that should be locked could possibly be in conflict with another cache line that is also locked, and thus lose its ability to be locked in the cache. For scratchpads, the basic allocation unit is only determined by the allocation scheme, so the contents cannot be polluted. However, if the basic allocation block is big, it is possible that the allocation block will not fit in the scratchpad at the end due to fragmentation.

Whitham and Audsley [117] introduced a hardware scratchpad memory management unit (SPMMU) that manages the transfers of data between memory and the data scratchpad to eliminate *pointer aliasing* and *pointer invalidation*. *Pointer aliasing* occurs when the same memory location is referenced using different names (pointers). *Pointer invalidation* occurs when an object in a memory location is moved out from that memory location. As a result, an alias that points to the object before the move, ends up pointing to an incorrect object. They propose to separate *logical addresses* (used by the program) and *physical addresses* (identifying where an object resides). THe SPMMU maintains a table mapping the logical address and physical address. Although the SPMMU resides in hardware, its contents are controlled by software via explicit OPEN and CLOSE commands in the code. The user specifies the base address for the object, the size of the object and the physical address at which the object is being loaded to. The SMMU then performs the transfer, and updates an internal table mapping the logical address to the new physical location of the object. This simplifies analysis because it eliminates the need for whole-pointer analysis in the program.

### 5.2.3 DRAM

DRAM cells leak charge and have to be refreshed periodically to retain their state. However, the refreshes of DRAMs stalls other DRAM accesses, and potentially closes DRAM rows, which require additional precharges to reopen them. This causes DRAMs to be unpredictable for real-time systems, as the DRAM refreshes are usually controlled in hardware. Bhat and Muller [19] tackled this specific issue of DRAM refreshes by scheduling burst refreshes. They accounted for the DRAM refresh requirements into the software, and schedule refresh tasks to handle DRAM refreshes predictably. Two implementations are provided. The first is a pure software implementation, and use RAS-only refreshes to manually refresh the DRAM rows during the refresh task. The second implementation uses a hybrid software-hardware solution, where the software initiates a hardware DRAM refresh. Depending on the application needs, each refresh can contain smaller bursts at the cost of scheduling more refreshes. By scheduling the DRAM refresh, other DRAM accesses are more predictable because no conflict will arise from refreshes.

Akesson et al. [7, 8, 6] introduced the Predator, a predictable SDRAM memory controller. Here,"predictable" provides a guaranteed maximum latency and minimum bandwidth to each client, independent of the behavior of each client. Standard DDR2 SDRAM memory controllers schedule the requests of the different components dynamically. Predicting the execution time of a particular

component in such a system is difficult, because of interference on the shared DRAM resource. Predator is a hybrid between static and dynamic memory controllers. Predator precomputes a set of of read and write groups with corresponding static sequences of SDRAM commands. These static sequences allow the computing of latency bounds, and are scheduled by the backend dynamically. As predictor is meant to service multiple clients, requests by different clients are scheduled using a Credit-Controlled Static-Priority arbiter (CCSP). This provides a maximum latency and bandwidth to the clients based upon the guarantees of the backend. The front-end also may delay each response by the back-end up to its worst-case bound. This eliminates interactions between different requestors.

Paolieri et al. [78] presented the Analyzable Memory Controller (AMC), which uses a very similar approach to the Predator. The main difference between AMC and Predator is that the AM uses a Round-Robin (RR) arbiter, instead of a CCSP arbiter employed in Predator. The RR arbiter provides the same latency and bandwidth guarantees to all clients while the CCSP provides better latency guarantee for high priority tasks.

# Chapter 6

# Conclusion and Future work

## 6.1 Summary of Results

This is my summary

It is important to understand that we are not proclaiming that all dynamic behavior in systems are harmful. But only by achieving predictability in the hardware architecture can we begin to reason about more dynamic behavior in software. For example, we discussed that dynamically scheduling threads in hardware causes timing interference. However, it is not the switching of threads that is unpredictable, but how the thread switching is triggered that makes it predictable. For example, the Giotto [42] programming model specifies a periodic software execution model that can contain multiple program states. If such a programming model was implemented on a thread-interleaved pipeline, different program states might map different tasks to threads or have different number of threads executing within the pipeline. But by explicitly controller the thread switches in software, the execution time variances introduced is transparent at the software level, allowing potential for timing analysis.

## 6.2 Publications

## 6.3 Future Work

Here is what you can keep doing

Talk about future research challenges for a predictable architecture.

- synchronization of threads, atomic primitives and memory barrier?
- Bus and I/O architectures

# Bibliography


[1] Autosar (automotive open system architecture).

[2] Control data 6400/6500/6600 computer systems reference manual.

[3] GNU ARM Toolchains.

[4] O. Aciiçmez, Çetin Kaya Koç, and J.-P. Seifert. On the Power of Simple Branch Prediction Analysis. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, New York, NY, USA, 2007. ACM.

[5] O. Aclicmez, J. pierre Seifert, and C. K. Koc. Predicting secret keys via branch prediction. In *in Cryptology CT-RSA 2007, The Cryptographers Track at the RSA Conference 2007*, pages 225–242. Springer-Verlag, 2007.

[6] B. Akesson. *Predictable and Composable System-on-Chip Memory Controllers*. PhD thesis, Eindhoven University of Technology, Feb. 2010. ISBN: 978-90-386-2169-2.

[7] B. Akesson, K. Goossens, and M. Ringhofer. Predator: a predictable SDRAM memory controller. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 251–256, New York, NY, USA, 2007. ACM.

[8] B. Akesson, A. Hansson, and K. Goossens. Composable resource sharing based on latency-rate servers. In *Proc. DSD*, Aug. 2009.

[9] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens. Real-time scheduling using credit-controlled static-priority arbitration. In *RTCSA*, pages 3 –14, Aug. 2008.

[10] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '83, pages 177–189, New York, NY, USA, 1983. ACM.

[11] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (VISA): exceeding the complexity limit in safe real-time systems. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 350–361, New York, NY, USA, 2003. ACM.

[12] ARM. *ARM Architecture Reference Manual*. ARM, July 2005.

[13] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 1(1):6–26, 2002.

[14] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad Memory: A Design Alternative for Cache On-chip Memory in Embedded Systems. *Hardware/Software Co-Design, International Workshop on*, 0:73, 2002.

[15] S. Bandyopadhyay. Automated memory allocation of actor code and data buffer in heterochronous dataflow models to scratchpad memory. Master's thesis, University of California, Berkeley, August 2006.

[16] J. Barre, C. Rochange, and P. Sainrat. A predictable simultaneous multithreading scheme for hard real-time. In *ARCS'08: Proceedings of the 21st international conference on Architecture of computing systems*, pages 161–172, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] H. Bauer. *Diesel-Engine Management*. Society of Automotive Engineers, 3rd edition, 2004.

[18] D. J. Bernstein. Cache-timing Attacks on AES, 2004.

[19] B. Bhat and F. Mueller. Making DRAM refresh predictable. In *ECRTS '10: Proceedings of the 22nd Euromicro Conference on Real-Time Systems*, Washington, DC, USA, 2010. IEEE Computer Society.

[20] E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. *Lecture Notes in Computer Science*, 1294:513–525, 1997.

[21] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real time systems. In *ECRTS '05: Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 33–40, Washington, DC, USA, 2005. IEEE Computer Society.

[22] S. C. Bono, M. Green, A. Stubblefield, A. Juels, A. D. Rubin, and M. Szydlo. Security analysis of a cryptographically-enabled rfid device. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[23] D. Brumley and D. Boneh. Remote timing attacks are practical. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 1–1, Berkeley, CA, USA, 2003. USENIX Association.

[24] C. Burguiere, C. Rochange, and P. Sainrat. A case for static branch prediction in real-time systems. In *RTCSA '05: Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 33–38, Washington, DC, USA, 2005. IEEE Computer Society.

[25] W. C.B., R. Walter, G. Aviation, and G. Rapids. Transitioning from federated avionics architectures to integrated modular avionics. *26th Digital Avionic Conference*, October 2007.

[26] D. Chaum. Blind Signatures for Untraceable Payments. In *Advances in Cryptology: Proceedings of Crypto 82*, pages 199–203. Plenu Press, 1983.

[27] W. Cheung, W. Evans, and J. Moses. Predicated instructions for code compaction. In A. Krall, editor, *Software and Compilers for Embedded Systems*, volume 2826 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin / Heidelberg, 2003.

[28] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors, 2009.

[29] J.-F. Dhem, F. Koeune, P.-A. Leroux, P. Mestre, J.-J. Quisquater, and J.-L. Willems. A Practical Implementation of the Timing Attack. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of the Third Working Conference on Smart Card Research and Advanced Applications (CARDIS 1998)*. Springer-Verlag, 1998.

[30] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. pages 264–265, June 2007.

[31] A. El-Haj-Mahmoud, A. S. AL-Zawawi, A. Anantaraman, and E. Rotenberg. Virtual multiprocessor: an analyzable, high-performance architecture for real-time computing. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 213–224, New York, NY, USA, 2005. ACM.

[32] J. Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *RTAS '03: Proceedings of the The 9th IEEE Real-Time and Embedded Technology and Applications Symposium*, page 152, Washington, DC, USA, 2003. IEEE Computer Society.

[33] M. Feng, B. B. Zhu, M. Xu, S. Li, B. B. Zhu, M. Feng, B. B. Zhu, M. Xu, and S. Li. Efficient Comb Elliptic Curve Multiplication Methods Resistant to Power Analysis, 2005.

[34] Gaisler Research. LEON3 Implementation of the Sparc V8. Website: http://www.gaisler.com.

[35] Gamma Technologies. *GT-Suite Flow Theory Manual*, 7.1 edition.

[36] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence Estimation for Speculation Control. In *In 25th Annual International Symposium on Computer Architecture*, pages 122–131, 1998.

[37] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic Processing in Cell's Multicore Architecture. *IEEE Micro*, 26(2):10–24, 2006.

[38] R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 158–168, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[39] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM TODAES*, 14(1):1–24, 2009.

[40] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[41] J. L. Hennessey and D. A. Patterson. Computer architecture: A quantatative approach, forth edition. Forth Edition:Appendex G. page G–44, 2007.

[42] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84 – 99, jan 2003.

[43] T. A. Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366, issue 1881:3727–3736, 2008.

[44] S. Hily and A. Seznec. Out-of-order execution may not be cost-effective on processors featuring simultaneous multithreading. In *HPCA '99: Proceedings of the 5th International Symposium on High Performance Computer Architecture*, page 64, Washington, DC, USA, 1999. IEEE Computer Society.

[45] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 4096, pages 449–458, Seoul, Korea, Aug. 2006.

[46] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk.* Morgan Kaufmann Publishers, September 2007.

[47] JEDEC. *DDR2 SDRAM SPECIFICATION JESD79-2E.*, 2008.

[48] R. Karri, K. Wu, P. Mishra, and Y. Kim. Fault-Based Side-Channel Cryptanalysis Tolerant Rijndael Symmetric Block Cipher Architecture. In *DFT '01: Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, page 427, Washington, DC, USA, 2001. IEEE Computer Society.

[49] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. Side Channel Cryptanalysis of Product Ciphers. In *Journal of Computer Security*, pages 97–110. Springer-Verlag, 1998.

[50] R. Kirner and M. Schoeberl. Modeling the function cache for worst-case execution time analysis. In *DAC '07: Proceedings of the 44th annual Design Automation Conference*, pages 471–476, New York, NY, USA, 2007. ACM.

[51] P. Kocher, J. J. E, and B. Jun. Differential Power Analysis. In *Lecture Notes in Computer Science*, pages 388–397. Springer-Verlag, 1999.

[52] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Lecture Notes in Computer Science*, pages 104–113. Springer-Verlag, 1996.

[53] O. Kömmerling and M. G. Kuhn. Design Principles for Tamper-Resistant Smartcard Processors. In *USENIX Workshop on Smartcard Technology proceedings*, pages 9–20, 1999.

[54] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded java microcontroller. *Microprocessors and Microsystems*, 27:19–31, 2003.

[55] J. Kreuzinger, A. Schulz, M. Pfeffer, T. Ungerer, U. Brinkschulte, and C. Krakowski. Real-time scheduling on multithreaded processors. In *RTCSA '00: Proceedings of the Seventh International Conference on Real-Time Systems and Applications*, page 155, Washington, DC, USA, 2000. IEEE Computer Society.

[56] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *In Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, 1991.

[57] E. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[58] E. Lee and D. Messerschmitt. Pipeline interleaved programmable dsp's: Architecture. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 35(9):1320–1333, 1987.

[59] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[60] E. A. Lee. Absolutely positively on time: What would it take? *Computer*, 38:85–87, 2005.

[61] E. A. Lee. Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, May 2008. Invited Paper.

[62] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable Programming on a Precision Timed Architecture. In *CASES '08: Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, pages 137–146, New York, NY, USA, 2008. ACM.

[63] T. Lundqvist and P. Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[64] M. Lv, W. Yi, N. Guan, and G. Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 339–349, Washington, DC, USA, 2010. IEEE Computer Society.

[65] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the 25th annual international symposium on Microarchitecture*, MICRO 25, pages 45–54, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[66] S. McFarling. Combining branch predictors. *Digital Western Research Laboratory*, June 1993.

[67] H. McGhan and M. O'Connor. Picojava: A direct execution engine for java bytecode. *Computer*, 31(10):22–30, 1998.

[68] T. S. Messerges, E. A. Dabbish, and R. H. Sloan. Investigations of Power Analysis Attacks on Smartcards. In *In USENIX Workshop on Smartcard Technology*, pages 151–162, 1999.

[69] S. Metzlaff, S. Uhrig, J. Mische, and T. Ungerer. Predictable dynamic instruction scratchpad for simultaneous multithreaded processors. In *MEDEA '08: Proceedings of the 9th workshop on MEmory performance*, pages 38–45, New York, NY, USA, 2008. ACM.

[70] Micron Technology, Inc. Various methods of dram refresh – rev. 2/99, 1994. http://download.micron.com/pdf/technotes/DT30.pdf.

[71] J. Mische, S. Uhrig, F. Kluge, and T. Ungerer. Exploiting spare resources of in-order smt processors executing hard real-time threads. In *ICCD*, pages 371–376, 2008.

[72] T. Mitra and A. Roychoudhury. A framework to model branch prediction for worst case execution time analysis. 2nd Workshop on WCET Analysis, Obtober 2002.

[73] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *In Cryptology ePrint Archive, Report 2005/368*, 2005.

[74] J. A. Muir. Techniques of side channel cryptanalysis. Master's thesis, University of Waterloo, 2001.

[75] National Institute of Standards and Technology. "Digital Signature Standard". Federal Information Processing Standards Publication 186, 1994.

[76] NVIDIA. Technical Breif: NVIDIA GeForce 8800 GPU Architecture Overview. Technical report, NVIDIA, Santa Clara, California, Nov 2006.

[77] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. From a federated to an integrated automotive architecture. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 28(7):956–965, 2009.

[78] M. Paolieri, E. Quinones, F. Cazorla, and M. Valero. An analyzable memory controller for hard real-time CMPs. *Embedded Systems Letters, IEEE*, 1(4):86 –90, dec. 2009.

[79] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee. A Timing Requirements-Aware Scratchpad Memory Allocation Scheme for a Precision Timed Architecture. Technical Report UCB/EECS-2008-115, EECS Department, University of California, Berkeley, Sep 2008.

[80] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. Technical Report UCB/EECS-2008-115, EECS Department, University of California, Berkeley, Sep 2008.

[81] C. Percival. Cache missing for fun and profit. In *Proc. of BSDCan 2005*, page 05, 2005.

[82] C. Percival. Hyper-threading considered harmful. http://www.daemonology.net/hyperthreading-considered-harmful/, 2005.

[83] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*, page 114, Washington, DC, USA, 2002. IEEE Computer Society.

[84] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[85] P. Puschner and A. Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, WORDS '02, pages 85–, Washington, DC, USA, 2002. IEEE Computer Society.

[86] J. W. Ramsey. Integrated modular avionics: Less is more. Avionics Magazine, February 2007.

[87] Red Hat. Red Hat Certificate System 7.3, Administration guide, B2. Encryption and Decryption.

[88] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Syst.*, 37(2):99–122, 2007.

[89] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. Pret dram controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS '11: Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 99–108. ACM, October 2011.

[90] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. In *WCET*, 2006.

[91] C. Rochange and P. Sainrat. A time-predictable execution mode for superscalar pipelines with instruction prescheduling. In *CF '05: Proceedings of the 2nd conference on Computing frontiers*, pages 307–314, New York, NY, USA, 2005. ACM.

[92] A. Sangiovanni-Vincentelli and M. D. Natale. Embedded system design for automotive applications. *Computer*, 40:42–51, 2007.

[93] T. U. Sascha Uhrig, Stefan Maier. Toward a processor core for real-time capable autonomic systems. In *Proceedings of the Fifth IEEE International Symposium on Signal Processing and Information Technology*, 2005.

[94] P. Schaumont, K. Sakiyama, Y. Fan, D. Hwang, S. Yang, A. Hodjat, B. Lai, and I. Verbauwhede. Testing ThumbPod: Softcore bugs are hard to find. In *Eighth IEEE International High-Level Design Validation and Test Workshop, 2003*, pages 77–82, 2003.

[95] M. Schoeberl. A time predictable instruction cache for a java processor. In *OTM Workshops*, pages 371–382, 2004.

[96] M. Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*. IEEE, 2005.

[97] M. Schoeberl. A time predictable java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, 2006.

[98] M. Schoeberl. A java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1-2):265 – 286, 2008.

[99] M. Sellnau, J. Sinnamon, L. Oberdier, C. Dase, M. Viele, K. Quillen, J. Silverstri, and I. Papadimitriou. Development of a practical tool for residual gas estimation in ic engines. In *SAE Paper 2009-01-0695*, 2009.

[100] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990.

[101] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[102] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. *ACM Trans. Program. Lang. Syst.*, 32(4):13:1–13:47, Apr. 2010.

[103] M. E. Tat and J. H. V. Gerpen. Measurment of biodiesel speed of sound and its impact on injection timing. Technical report, Dempartment of Mechanical Engineering, Iowa State University, 2003. Prepared under NREL subconract ACG-8-18066-01 for the National Renewable Energy Laboritory.

[104] L. Thiele and R. Wilhelm. Design for Timing Predictability. *Real-Time Systems*, 28(2):157–177, 2004.

[105] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, New York, NY, USA, 1995. ACM.

[106] S. Udayakumaran and R. Barua. Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In *CASES '03: Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 276–286, New York, NY, USA, 2003. ACM.

[107] T. Ungerer et al. MERASA: Multi-core execution of hard real-time applications supporting analysability. *IEEE Micro*, 99, 2010.

[108] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35:29–63, March 2003.

[109] M. Viele, I. Liu, G. Wang, H. Andrade, and B. Wilson. Remote sensing of fuel systems using real-time 1d cfd. ASME, To appear in ICES, 2012.

[110] D. W. Wall. Limits of instruction-level parallelism. *SIGARCH Comput. Archit. News*, 19(2):176–188, Apr. 1991.

[111] Z. Wang and R. B. Lee. Covert and Side Channels Due to Processor Architecture. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 473–482, Washington, DC, USA, 2006. IEEE Computer Society.

[112] Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on Computer architecture*, pages 494 – 505, San Diego, CA, June 2007 2007.

[113] L. Wehmeyer and P. Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *DATE*, pages 600–605, 2005.

[114] J. Whitham and N. Audsley. MCGREP - A Predictable Architecture for Embedded Real-time Systems. In *Proc. RTSS*, pages 13–24, 2006.

[115] J. Whitham and N. Audsley. Forming virtual traces for wcet analysis and reduction. pages 377–386, 2008.

[116] J. Whitham and N. Audsley. Predictable out-of-order execution using virtual traces. In *Proc. RTSS*, pages 445–455, 2008.

[117] J. Whitham and N. Audsley. Implementing time-predictable load and store operations. In *Proc. EMSOFT*, pages 265–274, 2009.

[118] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):1–53, 2008.

[119] R. Wilhelm et al. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE TCAD*, 28(7):966–978, 2009.

[120] E. Winward, J. Deng, and R. K. Stobart. Innovations in experimental techniques for the development of fuel path control in diesel engines. *SAE International Journal of Fuels and Lubricants*, 3(1):594–613, 2010.

[121] E. B. Wylie and V. L. Streeter. *Fluid transients*. McGraw-Hill, 1978.

[122] Xilinx. *Core generator guide*.

[123] Xilinx. *Xilinx Virtex-6 Family Overview*, March 2011.

[124] J. Yan and W. Zhang. A time-predictable VLIW processor and its compiler support. *Real-Time Systems*, 38(1):67–84, 2008.

[125] B. Ylvisaker, B. V. Essen, and C. Ebeling. A type architecture for hybrid micro-parallel computers. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:99–110, 2006.

[126] Yongbin. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.