Precision Timed Machines

Isaac Liu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2012-113
http://www.eecs.berkeley.edu/Pubs/TechRpts/2012/EECS-2012-113.html

May 14, 2012

Acknowledgement

I want to thank my wife, for supporting me throughout my Ph.D career and giving me the strength to complete this work. I want to thank my parents, for raising me with the values of perseverance and hard work.
I want to thank my advisor, Edward A. Lee, for always guiding and encouraging me. I want to thank all worked on the PRET project with me: Ben Lickly, Hiren Patel, Jan Rieneke, Sungjun Kim, Stephen Edwards, David Broman and Mike Zimmer. This work would not have been possible without you guys.
I want to thank the ptolemy group, especially Christopher and Mary, for taking care of everything so I could focus on my research.
I want to thank the committee members, for giving me wonderful feedback.

Finally, I want to thank everyone else that helped make this thesis possible.

Precision Timed Machines

by

Isaac Suyu Liu

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

in

Electrical Engineering and Computer Sciences

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

Precision Timed Machines

Abstract

Precision Timed Machines

by

Isaac Suyu Liu
Doctor of Philosophy in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Edward A. Lee, Chair

Cyber-Physical Systems (CPS) are integrations of computation with physical processes [65]. These systems must be equipped to handle the inherent concurrency and inexorable passage of time of physical processes. Traditional computing abstractions only concern themselves with the functional aspects of a program, and not its timing properties. Thus, nearly every abstraction layer has failed to incorporate *time* into its semantics; the passage of time is merely a consequence of the implementation. When the temporal properties of the system must be guaranteed, designers must reach beneath the abstraction layers. This not only increases the design complexity and effort, but the systems are overdesigned, brittle and extremely sensitive to change.

In this work, we address the difficulties of handling *time* in computing systems by re-examining the lower levels of abstraction. In particular, we focus on the instruction set architecture (ISA) layer and its affects on microarchitecture design. The ISA defines the contract between software instructions and hardware implementations. Modern ISAs do not constrain timing properties of instructions as part of the contract. Thus, architecture designs have largely implemented techniques that improve average performance at the expense of execution time variability. This leads to imprecise WCET bounds that limit the timing predictability and timing composability of architectures.

In order to address the lack of temporal semantics in the ISA, we propose instruction extensions to the ISA that give temporal meaning to the program. The instruction extensions allow programs to specify execution time properties in software that must be observed for any *correct* execution of the program. These include the ability to specify a minimum execution time for code blocks, and the ability to detect and handle missed deadlines from code blocks that exhibit variable execution times. This brings control over timing to the software and allows programs to contain timing properties that are independent of the underlying architecture. In addition, we present the Precision Timed ARM (PTARM) architecture, a realization of Precision Timed (PRET) machines [32] that provides timing predictability and composability without sacrificing performance. PTARM employs a predictable thread-interleaved pipeline with an exposed memory hierarchy that uses scratchpads and a predictable DRAM controller. This removes timing interference among the hardware threads, enabling timing composability in the architecture, and provides deterministic execution times for instructions within the architecture, enabling timing predictability in the architecture. We show that the predictable thread-interleaved pipeline and DRAM controller design also achieve better throughput compared to conventional architectures when fully utilized, accomplish-

ing our goal to provide both predictability and performance.

To show the applicability of the architecture, we present two applications implemented with the PRET architecture that utilize the predictable execution time and the extended ISA to achieve their design requirements. The first application is a real-time fuel rail simulator that implements a one dimensional computational fluid dynamics (1D-CFD) solver on a multicore PRET architecture. The implementation leverages the timing instructions to synchronize the communication of multiple PRET cores with low overhead. The predictable nature and the improved throughput of the architecture allow us to optimize the resource usage while statically ensuring that the timing requirements are met. This provides a scalable solution to close the loop of fuel delivery, allowing for more precise fuel injections that lead to a cleaner and more efficient engine. The second application presents a case study that uses PRET to remove the vulnerability of timing side-channel attacks on encryption algorithms. Encryption algorithms are vulnerable to side-channel attacks that measure the execution time of the encryption to derive the encryption key. The uncontrollable execution time variance can stem from the unpredictable sharing of architecture features or from the various control paths of the encryption algorithm. We implement the RSA and DSA [82] encryption algorithms on PRET and show that by using the timing extended ISA and a predictable architecture, we can completely remove the vulnerabilities that are exploited for the attacks.

By providing a predictable architecture, we provide simpler and more accurate timing analysis of the software. With the instruction extensions to the ISA, we provide timing control and allow architecture independent timing properties to be specified in the software. Through these contributions, we aim to introduce a timing deterministic foundation to the lower levels of computing abstractions, which enables more precise and efficient control over timing for the design of CPS.

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, for supporting me throughout my Ph.D career, and giving me the strength to complete this work.

I want to thank my parents, for raising me and teaching me the values of perseverance and hard work.

I want to thank my advisor, Edward A. Lee, for always guiding and encouraging me.

I want to thank all the collaborators that worked on the PRET project with me: Ben Lickly, Hiren Patel, Jan Rieneke, Sungjun Kim, Stephen Edwards, David Broman and Mike Zimmer. This work would not have been possible without you guys.

I want to thank the whole ptolemy group, especially Christopher and Mary, for always taking care of everything so I could focus on my research.

I want to thank Jia, for providing me the template and valuable experience for the thesis writing process.

I want to thank the committee members, for giving me wonderful feedback on my thesis.

Finally, I want to thank everyone else not mentioned here that helped make this thesis possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Cyber-Physical Systems (CPS) are integrations of computation with physical processes [65]. 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 number of applications can be categorized as 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 their underlying tasks to deliver services in real time. The timing constraints of *soft real-time systems* are typically used to 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 developed 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 place emphasis on improving average performance over predictability. As a result, when computing systems absolutely must meet tight timing constraints, these computing advances often do more harm than good [64]. The scale and complexity of traditional embedded systems allow designers to compensate with extra effort in design and analysis. However, these solutions begin to break down when transitioning to larger scale 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 timing semantics 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 can 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 [61]. 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 [112].

- *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 [112]. Furthermore, because the large variation of execution time in modern complex processors, worst-case execution time (WCET) analysis techniques often lead to overly conservative results for safety [125]. 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 [99, 32]. For example, Sangiovanni-Vincentelli et al.[99] show 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 small program additions. 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 about and prove the timing properties of the full system. For avionics manufacturers, 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. [112], Henzinger [46] and Lee [64] 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* [46]. If the definition of *output* includes the timing behavior exhibited by the system, then current abstractions 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 are 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 be equipped to handle the *inherent concurrency* and the *inexorable passage of time* for physical processes. Sangiovanni-Vincentelli et al. [99] identified these issues as the *timing composability* and *timing predictability* of systems, and list 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 the platforms are feature specific, they are often idle during run time. In order to reduce resource consumption, there is a shift towards *integrated architectures* [84, 27], where multiple functions are integrated on a single, shared platform. Several challenges exists to make the 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 [93] 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 that 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 abstraction that have been adopted from conventional computing advances must be redefined to allow for timing 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 the timing properties of the system. Timing composition plays a big part of this when features are integrated, but even separately, it is difficult to analyze the execution time of programs.

Wilhelm et al. [125] describe the abundance of research and effort that has been put into bounding the WCET of programs. Not only is determining the worst case program flow a challenge, but the precision and usefulness of the analysis also depend on the underlying architecture [43]. Conventional architectures have implemented techniques that target the improvement of average case execution time (ACET) at the expense of execution time variability. As a result, it is 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-provisioning to ensure timing requirements are met. Thus, time determinism and reduced jitter are needed for future systems to increase performance [99].

As modern layers of abstraction for computing 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 of the same family. Thus, the redefinition of abstraction layers must also include temporal semantics to allow reasoning of timing properties at each layer independently from the abstract layers below it.

1.2 Contributions

The contribution of this work is to provide more precise and efficient control over the timing properties of computing systems. Specifically in this thesis, we focus on the lack of temporal semantics in 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 the architecture. The key challenges we contribute to are twofold.

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 archi-

tectures. PREcision Timed (PRET) machines [32] are designed with *timing-predictability* as the main metric for success. However, predictability can be easily achieved if one is willing to forgo performance; computer architectures 50 years ago were very predictable. Thus, our contribution is to deliver both predictability and performance. 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 good. We do not intend to reinvent computing advancements, but instead evaluate them through the lenses of predictability and composability. In doing so, we present an architecture designed for timing-predictability without sacrificing 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 can easily thwart architecture innovation opportunities. Thus, we explore 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 both contributions are essential to cyber-physical systems. Without a predictable architecture, programs can exhibit unpredictable execution time variances. But 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. With both the ISA extensions and the predictable architecture, we equip platforms with the ability to interact with physical processes and provide a solid foundation to enable precise and efficient control over the timing properties of the system. This prevents the overdesigning of computing systems for applications that absolutely must meet timing constraints, such as CPS.

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

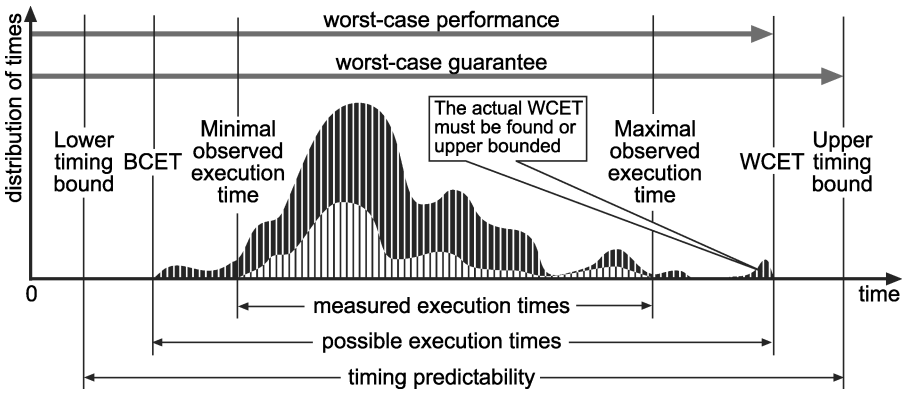


Figure 1.1: “Program Execution Times [125]”

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 best-case execution time (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 [125]. *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 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 architecture's 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. The outcome of the branch predictor could easily affect the execution time of the program. Superscalar architectures can execute instructions out-of-order, so 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.

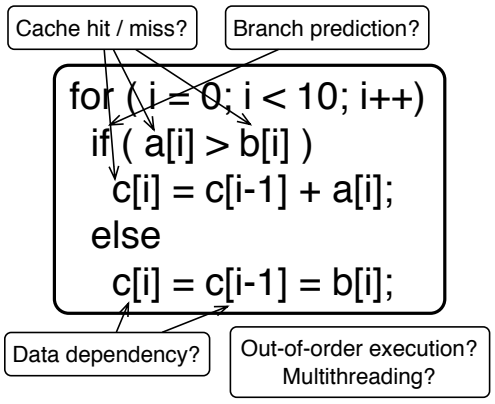


Figure 1.2: Simple Loop Timing Issues

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 pro-

cesses 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 [113] 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* [97, 69]; situations where a local worst-case does not result in the global worst-case. Reineke et al. [97] illustrate this with the example shown 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.

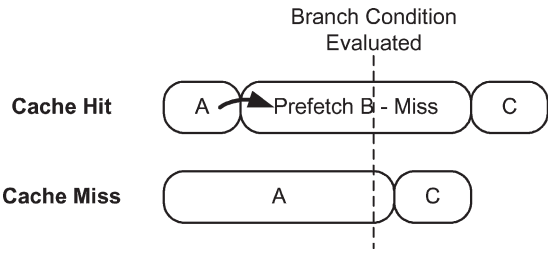


Figure 1.3: Timing anomaly cause by speculation [97]

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 [43]. 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 present the design and implementation of a Precision Timed (PRET) machine [32] – the Precision Timed ARM (PTARM). PTARM employs a thread-interleaved pipeline and a memory controller designed for predictable and composable execution. It also implements an extended ARM [15] ISA to demonstrate the ISA extensions with temporal semantics. Our benchmarks show that an architecture designed for timing predictability and composability does not need to sacrifice performance.

Many people have contributed to the results given in this thesis. The predictable DRAM controller that is presented in section 2.2.2 is a collaborative effort jointly done with Jan Reineke and Sungjun Kim. Hiren Patel, Ben Lickly, Jan Reineke, David Broman and Edward Lee have all greatly contributed to timing extensions to the ISA presented in section 2.3. And finally, the engine fuel rail simulation application presented in section 4.1 is a collaborated effort with Matthew Viele, Guoqiang Wang and Hugo Andrade. It is a pleasure to thank those who made this thesis possible, as this thesis could not have been complete without them.

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 tradeoffs when designing architectures for predictable and repeatable timing.

2.1 Pipelines

The introduction of pipelining vastly improves 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 in each processor cycle, one instruction completes and leaves the pipeline as another enters and begins execution. In reality, different pipeline hazards occur that 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 are not yet available. Pipelines begin the execution of instructions before preceding ones are finished. Thus, consecutive instructions that are data-dependent can simultaneously be executing in the pipeline. For example, the code in figure 2.1 shows assembly instructions from the ARM instruction

```
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

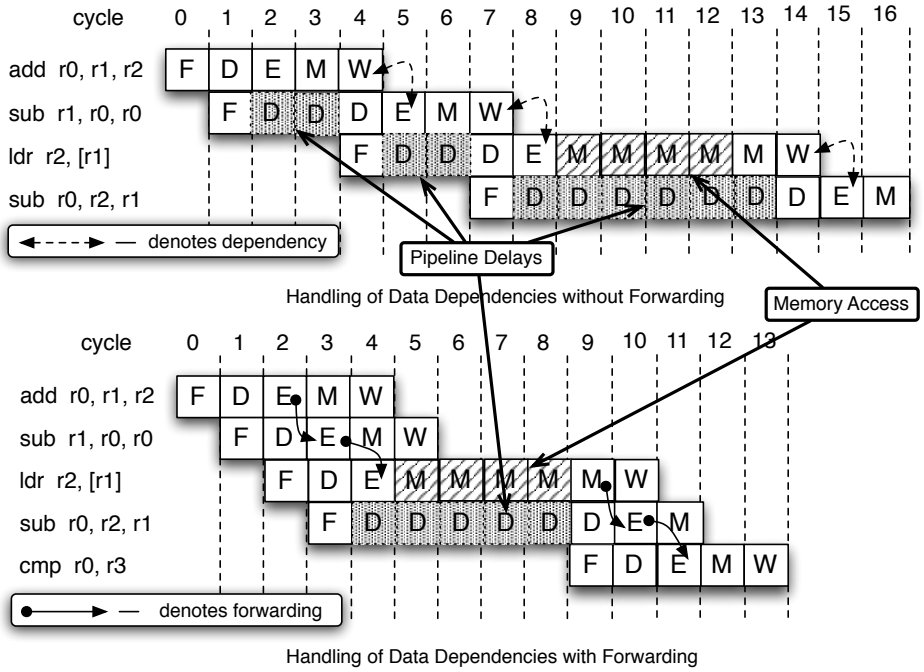


Figure 2.2: Handling of data dependencies in single threaded pipelines

set architecture (ISA). Each instruction in the code segment depends 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 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 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 delays inserted in the pipeline. The circuitry for 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.

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. 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 whether any stalls are inserted. The length of the stall is determined by the execution time of the dependent instructions, because the pipeline will stall until those instructions complete. 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.

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

Control Hazards

Branches cause 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 complicate matters, as whether or not the branches are taken depends on an additional condition that could also be unknown when the conditional branches are 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. The conditional bits can be set based on the results of standard arithmetic instructions [15]. 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

```
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

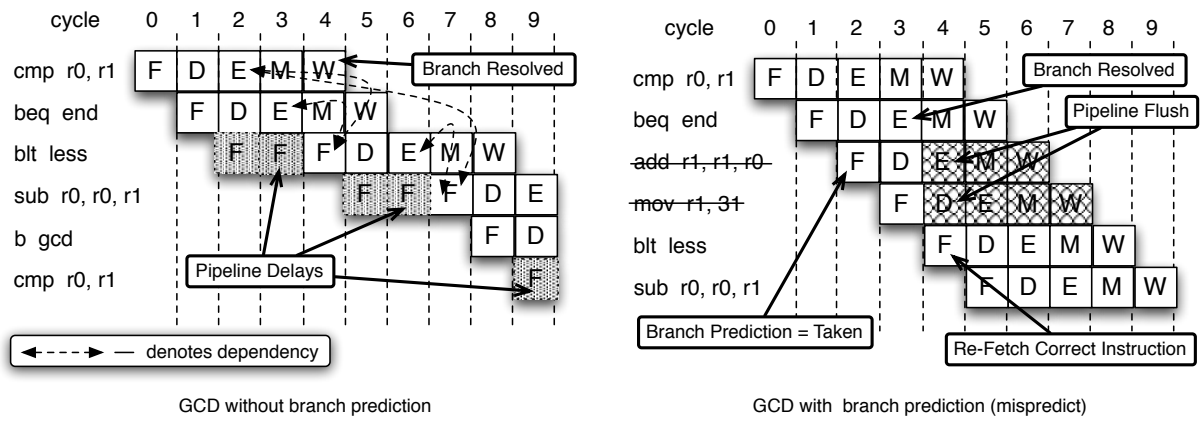


Figure 2.4: Handling of conditional branches in single threaded pipelines

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 same 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 require 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 its execution. Branch predictors internally maintain a 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% [73]. If the branch prediction is correct, no penalty is incurred for the branch because the correct instructions are speculatively executed. However, when the prediction is incorrect (often referred to 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* instructions 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.

Stalling and branch predicting 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 needs 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, 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. However, 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 [79], 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 are used, their internal state could be updated during the speculative execution of a mispredicted path. As architectures grow in complexity, the combined modeling of all hardware states in the architecture often leads to an intractable 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 highlights an important tradeoff 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 decreasing of the worst-case performance. For real-time and safety critical systems, the challenge remains to improve worst-case performance while maintaining predictability, and how pipeline hazards are handled plays a key role in tackling this challenge.

Although less often mentioned, the presence of interrupts and exceptions in the pipeline also creates control hazards. Exceptions can occur during the execution of any instruction and change 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. 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 enforce sequential access to the contended hardware component. 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 register file and program counter. 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* refers to the explicit hardware threads that have physical 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 al. [115] survey different multithreaded architectures and categorize 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 similarly 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 fewer threads

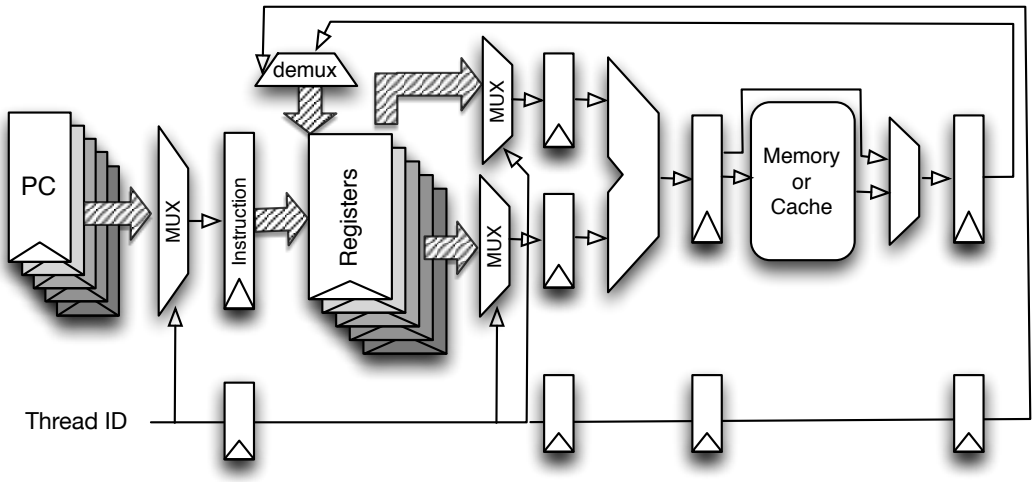


Figure 2.5: Simple Multithreaded Pipeline

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 frequently as every processor cycle. The *execution width* of the pipeline is 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 [113] is an example where multiple instructions are fetched from different threads each cycle.

Multithreaded architectures present several challenges for static execution time analysis. As figure 2.5 illustrates, 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 times 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 needs to be known. The interleaving of threads depends 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* [117]. Conventional multithreaded 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 sev-

eral 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 CDC6600 [2]. Each "peripheral processor" is implemented as a hardware thread. Interacting with input/output peripherals often lead to idle processor cycles to wait 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.

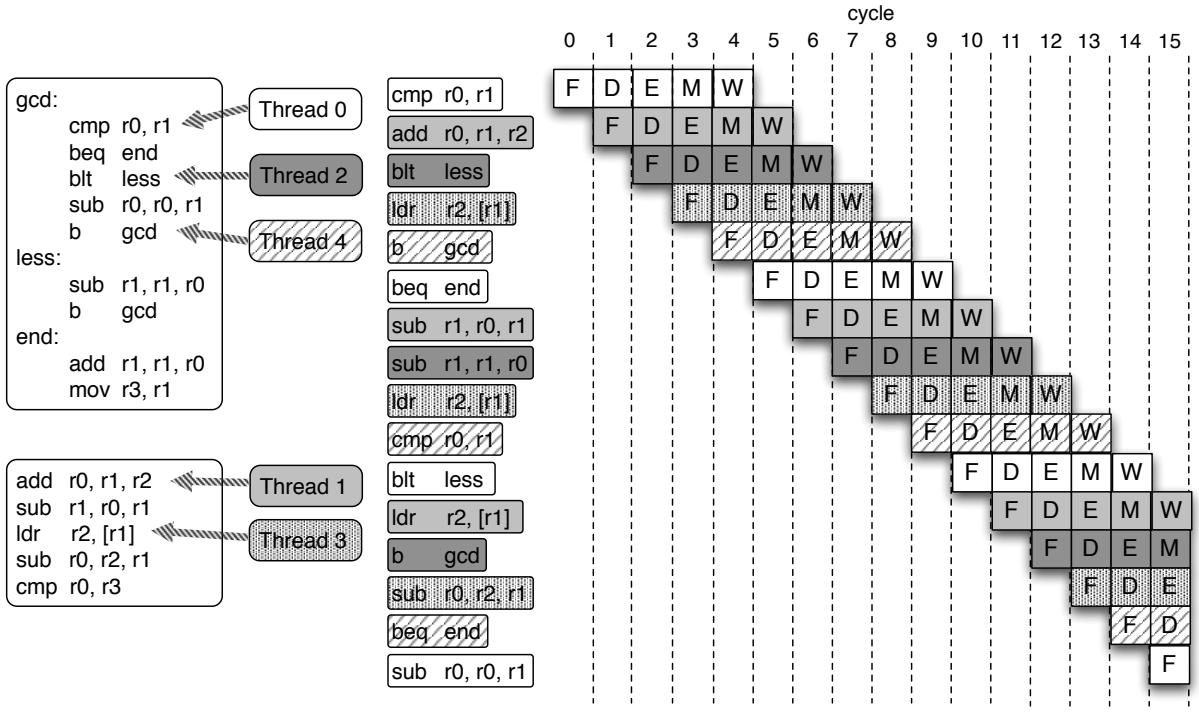


Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

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 progress 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 multi-threaded architectures to achieve predictable performance, threads must be temporally isolated from one another. Temporal isolation removes cross-thread timing dependencies to allow 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 eliminates 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 will be executing an instruction from a different hardware thread. Thus, no explicit dependency will exist between instructions in the pipeline. Lee and Messerschmitt [62] 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 pipelining 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

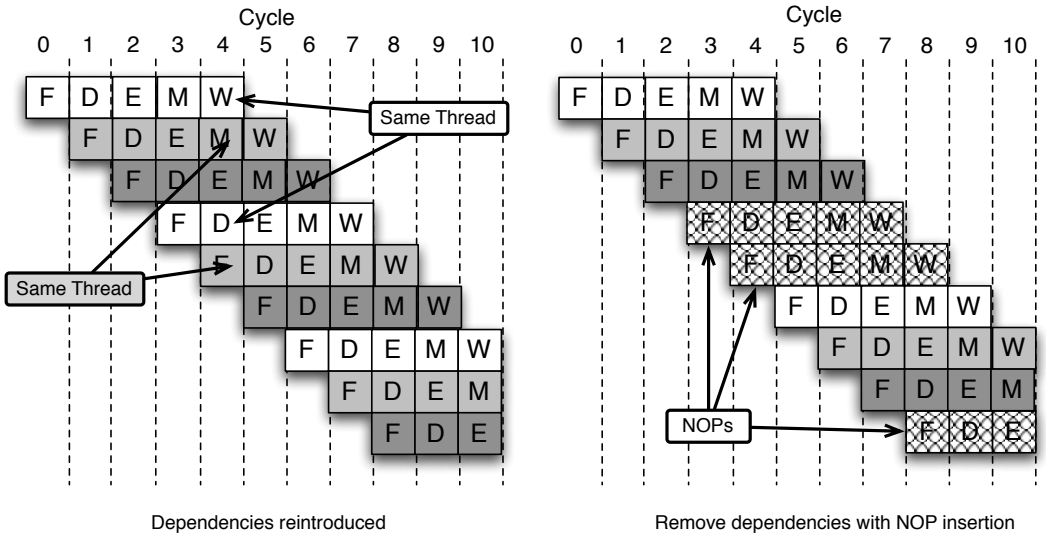


Figure 2.7: Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive

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 that 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 are still executing during the long latency operation. But the temporal isolation of threads is preserved, as the execution frequency of threads remains 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 expose 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 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 [70].

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 whether a memory access goes to the cache or to the 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 [119] 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 [53] 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. The minimum observable latency for instructions depend on the number of threads executing 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,

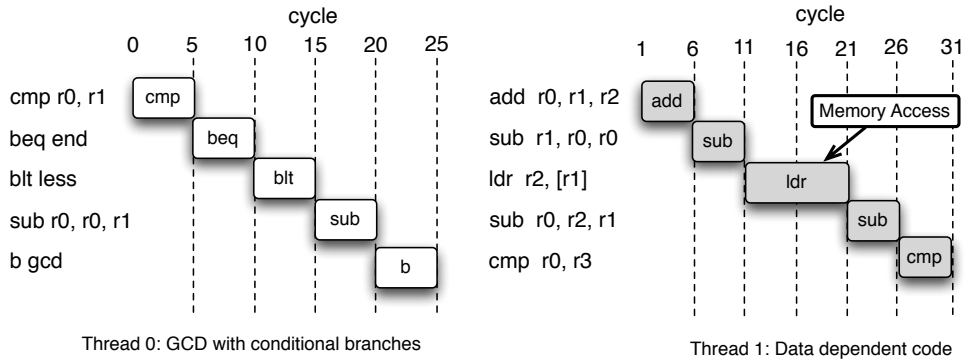


Figure 2.8: Isolated execution of threads with a thread-interleaved pipeline

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 in 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, each 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 bottleneck to improved memory performance. Common memory technologies used in embedded systems contain a significant tradeoff 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 from being implemented with high capacity. On the other hand, Dynamic Random-Access Memories (DRAM) use a more compact memory cell design that can easily be combined into larger capacity memory blocks. But the memory cell of DRAMs must be constantly refreshed due to charge leakage, and the large capacity memory blocks often prohibit 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 and placing less accessed values 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 the 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 are routed through the cache. If the data from the memory access is in the cache, then a cache hit occurs, and the data is returned right away. However, if data is not in the cache, then a cache miss occurs, and the cache controller fetches the data from the larger memory and adjusts the memory contents in 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 tradeoff 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 update 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.

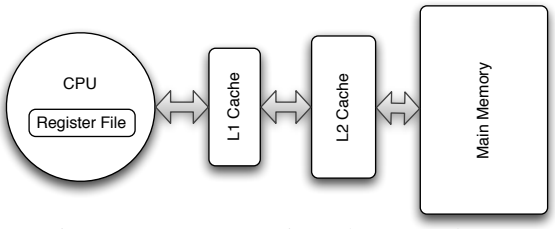


Figure 2.9: Memory Hierarchy w/ Caches

When caches are used, the program is oblivious to the different levels of memory hierarchy because they are 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 they provide.

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 with 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 with program execution to predict the 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 [43]. The N different memory accesses will evict all cache lines in the cache prior to the basic block, and fill them 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 unintended side effects. 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 tuned and tailored to specific cache architectures and parameters to achieve the desired performance. In order to tune algorithm performance, algorithm designers are required to understand the abstracted away memory architecture and enforce data access patterns that conform to the cache size and replacement 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* [60], and is very well-known and commonly used.

Multithreaded threaded architectures with shared caches among 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 [108] 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 [16]. Scratchpads can be found in the Cell processor [39], which is used in Sony PlayStation 3 consoles,

and NVIDIA's 8800 GPU, which provides 16KB of SPM per thread-bundle [83]. Scratchpads use the same memory technology (SRAMs) as caches, but do not implement the hardware controller to manage their memory contents. 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 [16].

Unlike caches, which overlay their address space with main memory to hide the hierarchy, scratchpads explicitly *expose* 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 removes the variability of execution time introduced with caches. However, this places the burden of memory management on the programmer or compiler toolchains. The Cell processor [39] is often criticized for being difficult to program, and one of the main reason is its use of scratchpads. Programmers have become accustomed to a uniform memory space, making it difficult to adjust to the non uniform memory space that must be explicitly managed.

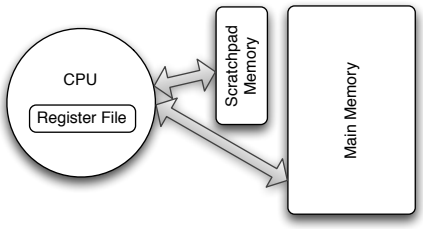


Figure 2.10: Memory Hierarchy w/ Scratchpads

Embedded system designs inherently need to deal with limited resources and other design constraints, such as limited memory or hard timing deadlines. Thus, the design of such systems often requires analysis of memory usage and latency to ensure that the constraints are met. These analysis results can be used to generate automatic allocation schemes for scratchpads, lessening the burden on programmers. 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 do not change throughout program execution. Static scratchpad allocation schemes [109, 86] often use heuristics or a compiler-based static analysis of the program to find the most commonly executed instructions or data structures. These are allocated 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. Higher level models of computation, such as Synchronous Dataflow (SDF) [63] or Giotto [45], expose more structure and semantics of the model for better analysis, which can be used to optimize scratchpad allocation dynamically. Bandyopadhyay [17] presents 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 executes a SDF model that contains actors communicating with each other. Bandyopadhyay analyzes the actor code and the data that is communicated in each HDF state, and derives an optimized scratchpad allocation for each state. The scratchpad allocation code is automatically inserted

into the code to dynamically change the scratchpad contents during state transitions, so the memory allocation is optimized for the execution of each HDF state. This allocation not only shows roughly a 17% performance improvement compared to executions using LRU caches, but also a 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, by using caches in the memory hierarchy, the hierarchy is hidden from the programmer, and the hardware managed memory contents create highly variable execution times with unpredictable access latencies. Scratchpads on the other hand expose the memory hierarchy to the programmer, allowing for more predictable and repeatable memory access performances. Although the allocation of scratchpads requires more programming effort, 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 the 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 times. The contributions from this section are research done jointly with the several co-authors from Reineke et. al [96]. We do not claim sole credit for this work, and the summary is included in this thesis only for completeness. We will first give some basic background on DRAM memories, then present the predictable DRAM controller designed.

DRAM Basics

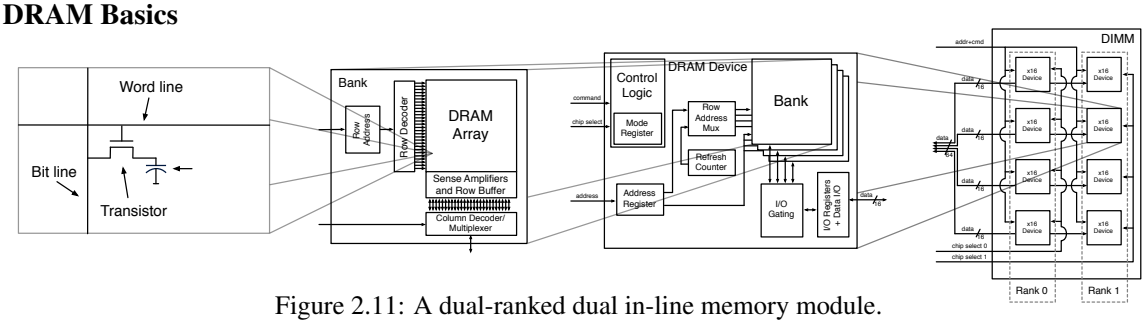


Figure 2.11: A dual-ranked dual in-line memory module.

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 [51].

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 is 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 figure 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. One common way is to issue a refresh command that 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 perform 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 used to determine which ranks are 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

[1] This table is as shown in [96]

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

(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 that would result in too high of a current draw. The memory controller must conform to these timing constraints when sending commands to the DDR2 module. Here we only give a quick overview of DRAMs, we refer more interested readers to Jacob et al. [50] 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. [96]

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 refreshes. Here we specifically refer to a DDR2 667MHz/PC2-5300 memory 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.

[3] In cycles at 200 MHz

Backend Conventional DRAM memory controllers view the entire memory device as one resource, and any memory request 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 resources 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 for 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 go 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.

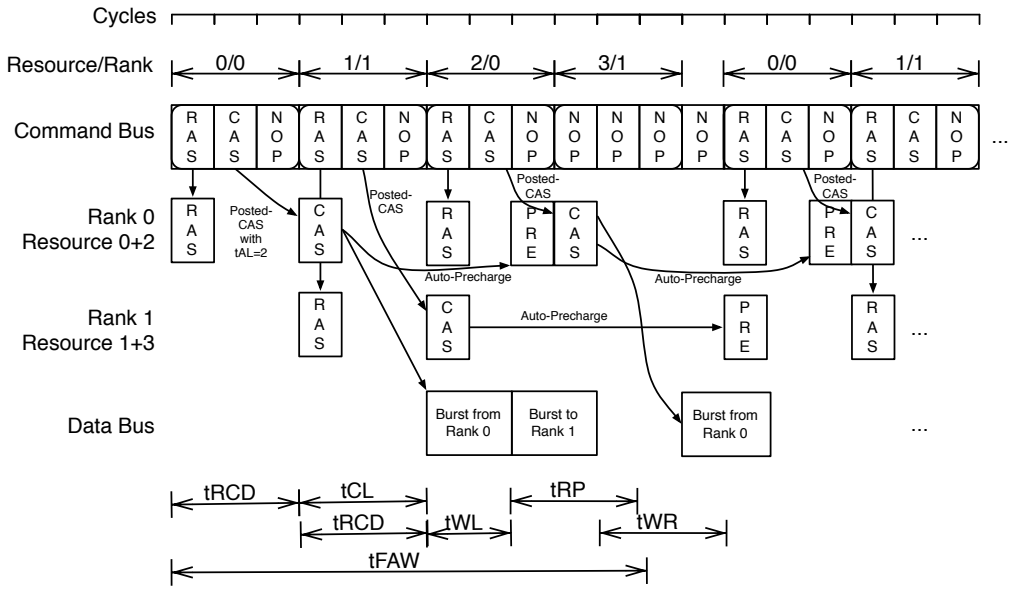


Figure 2.12: The periodic and pipelined access scheme employed by the backend [96].

Each access request is translated into a RAS (Row Access), posted-CAS (Column Access) and NOP command. An access slot is formed of all three commands. 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, 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 figure 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, and 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.

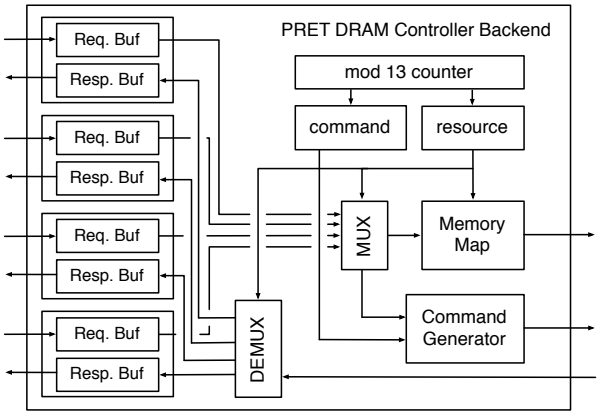


Figure 2.13: Sketch of implementation of the backend [96].

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 are 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, such as MERASA [114], PREDATOR [9], JOP [105], or CoMPSoC [42], strive to develop predictable multi-core architectures that 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 have 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 independent 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 [11] 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 [96].

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 require fewer 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 now need to account for potential refresh command latencies, which vary depending on the refresh progress. Instead, we use the distributed, RAS-only refresh [77] to each bank separately. Memory refreshes in

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

this case are equivalent to row accesses to a bank; each resource can be refreshed without effecting others. Manually accessing rows on the other give much shorter latencies each time, but incur a slight bandwidth hit because more accesses need to be performed to meet the refresh constraints. The shorter latencies however improve the worst-case access latency, because the refresh latency is shorter.

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. Conventionally, 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 allows us to completely hide memory refreshes in some unusable access slots lost in the synchronization. This provides 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) serves as the contract between the software and the hardware. The programmer understands the semantics of each instruction and uses it to construct programs. Computer architects ensure that the implementation of each instruction abides by the semantics specified in the ISA. The semantics of the instructions in modern ISAs often do not specify temporal properties for 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 properties 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 [45], 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 timing semantics of the ISA, 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 [49] propose a simple extension to the processor which implements the *deadline* instruction, an instruction that allows the programmer to specify a minimum execution time of code blocks. They show an implementation of a VGA controller in software by using the deadline instructions to control the output horizontal and vertical sync signals. Such functions are typically implemented in hardware because the timing precision required is hard to achieve in software. However, the deadline instruction provides the precise timing control needed, which enables a software implementation. We further expand on this concept of controlling execution time in software, and introduce a set of assembly timing instructions that allow us to control not only the minimum execution time, but to 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 for setting timing interrupts highly varies depending on platform implementation. The external timer is often viewed as just another I/O component, and access to the timer is often done through memory mapped registers. 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 of and possibilities for extending the 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 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 [66] 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 the 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 to control offsets of time. The actual value of the timestamps are 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 [138]. 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, 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 platform time.
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 [15] 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 that use our extensions.

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 them accessible to programmers. The programmer can manipulate the timestamps by using conventional data-processing instructions like add or subtract. However, because the timestamps are 64 bits, architectures with 32 bit registers store the value in 2 separate registers. Thus, any manipulation of timestamps must handle the overflow caused by 32 bits operations properly. Several ISAs provide an *add with carry* instruction that can be used, or else the programmer must explicitly do so in software. The timestamps are used as inputs to other timing instructions which we will introduce below.

We can technically implement the functionality of this instruction by memory mapping the platform clock to two 32 bit memory locations. This would be similar to conventional methods of accessing timers. However, loading a 64 bit time value would require 2 separate load instructions. 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 [49], 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 whether 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 ISA. The ARM ISA allows extra functionality to be added to the instruction set by providing 16 co-processor instruction spaces. 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 [49] 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

The *delay_until* instruction is only used to 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 that 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. This timeout value is checked in hardware as the program continues execution. 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*.

In the code sample, lines 1 to 3 are used to setup the timestamp passed into *exception_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

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
```

deadlines to be checked. Multiple timeout slots can be implemented and checked in hardware at the cost of hardware complexity.

Similar to *delay_until*, *exception_on_expire* and *deactivate_exception* merely create a mechanism to specify desired timing constraints. None of the timing instructions enforces 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 about 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 for static analysis to obtain a tight worst-case execution time bound.

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 these assembly level instructions can be used by higher level languages to synthesize different timing behaviors.

Constructing Different Timing Behaviors

Figure 2.14 shows four possible timing behaviors that we can construct for a code block using the assembly level timing instructions. The code block in this case 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 the task. 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.

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 with scenario A and B in figure 2.14. Scenario A is used if the execution of task needs to completed.

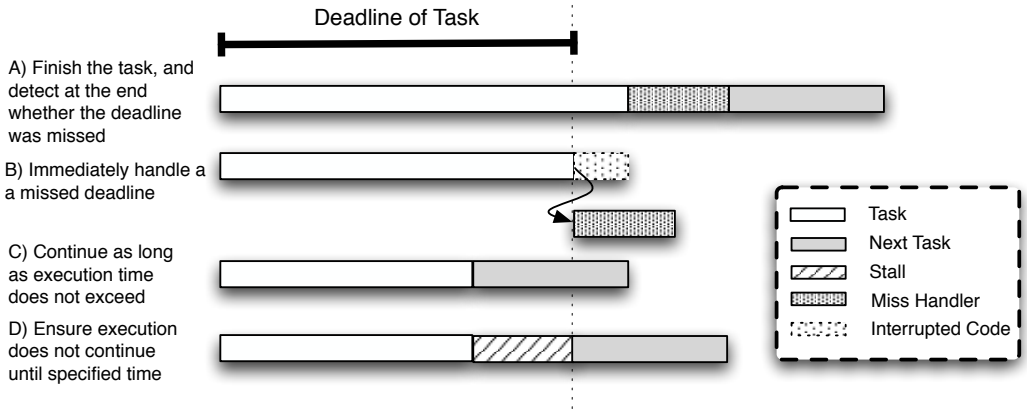


Figure 2.14: Different Desired Timing Behaviors

It could be that the task modifies external I/O states that cannot afford to be left in an unknown state. In this scenario, the task must first complete, then execute the miss handler before 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 are used to set up 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 are where the miss detection occur. 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* only occurs 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 immediately, 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 *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 task 1 to start task 2, or it could return to the program point where the exception was thrown after *miss_handler* completes. 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

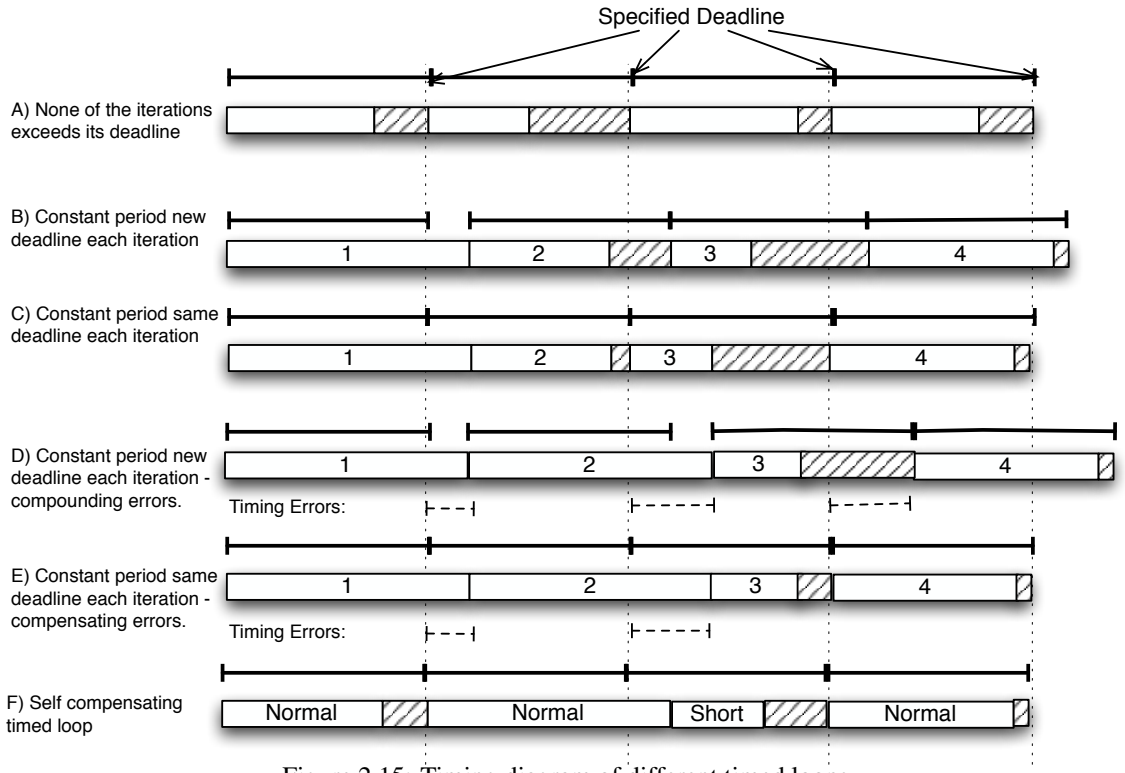


Figure 2.15: Timing diagram of different timed loops

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 lines 2 and 3 in listing 2.6. 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 different execution times, but the *delay_until* instruction ensures each iteration takes the

whole period to execute. However, if one iteration misses the deadline, and its execution time exceeds the period, then scenario B in figure 2.15 would be observed in 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 slightly modify 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 are purely the period of the loop, since we do not call *get_time* again to 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 both scenarios, 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 the execution time of iteration 3 is short enough to allow recovery. As shown in scenario E, our second implementation recovers the period by 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 the loop overhead and miss-detection is not accounted for, then we will always detect

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 r5, r3, r5             ; Check if previous iteration deadline is missed
8   sbc  r4, 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
```

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 whether the previous deadline was met, and lines 10 and 11 execute the short task if the deadline was missed, or execute the normal task otherwise. We assume that both tasks saves the processor condition codes in the preamble of the task, and restores it in the postamble. In this code, 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 exceed 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 with our timing extensions to point out the subtleties when doing so.

Chapter 3

Precision Timed ARM

In this chapter we introduce the Precision Timed ARM (PTARM). The PTARM is a realization of the PRET principles on the ARM instruction set architecture [15]. 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 easily analyzable and has repeatable timing.

Following the design principles discussed in chapter 2, PTARM employs a thread-interleaved pipeline and an exposed memory hierarchy with scratchpads and a timing predictable DRAM controller. The ARM ISA is chosen not only because of its popularity in the embedded community, but also because it is a *Reduced Instruction Set Computer* (RISC), which contains simpler instructions that allow for more precise timing analysis. *Complex Instruction Set Computers* (CSIC), such as Intel's x86 ISA, add 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 shift instructions. Instead, data-processing instructions can shift their 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 occurs when load/store instructions use the pre or post-index addressing mode. This is useful to compact code that operates 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 [29].

The 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. PTARM implements the ARMv4 ISA, without support for the thumb mode, an extension that compacts the instructions to 16 bits, instead of the typical 32 bits. 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. Curretly, PTARM is implemented as a *soft core* on the Xilinx Virtex-5 and 6 Family FPGAs, thus several design decisions were made to optimize PTARM for those FPGA families. Soft core processors are microprocessors that are synthesized onto FPGAs. They can often be customized with different feature sets and configurations before being implemented on the FPGA. The PTARM implements a 32 bit datapath with a five stage thread-interleaved pipeline. Thread-interleaved pipelines remove pipeline hazards 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 [62] show 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 to slightly improve thread latencies.

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. It contins 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 follows the five stage pipeline described in Hennessey and Patterson [44], 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 fetches the PC from different threads in a round robin fashion every cycle. A simple 2 bit up-counter is used to keep track of which thread 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, representing the time in nanoseconds, is associated with each instruction when it begins execution in the pipeline. This 64 bit timestamp is latched from the *timer* in the fetch stage, and is kept with the instruction for the duration of its execution.

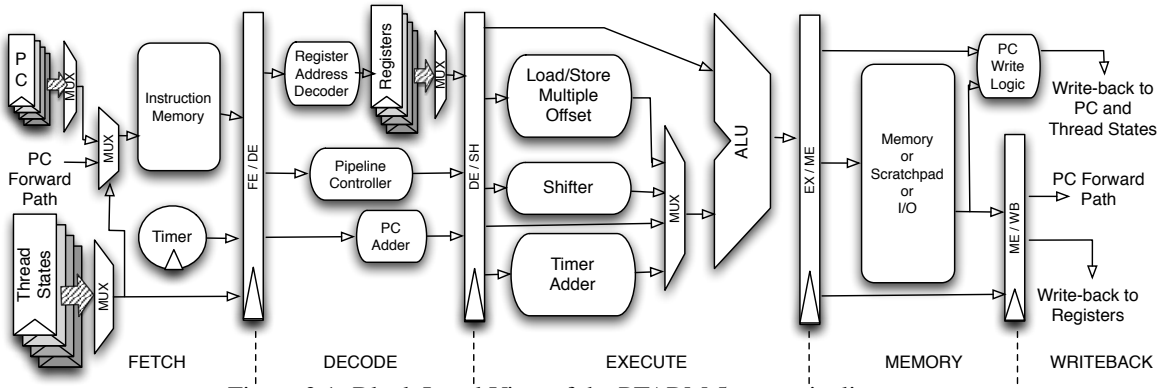


Figure 3.1: Block Level View of the PTARM 5 stage pipeline

The *decode stage* contains the *pipeline controller* that decodes instructions and determines the pipeline control signals to be propagated down the pipeline. Most of the 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. Conventional pipeline controllers need to keep track of all instructions currently executing in the pipeline, to detect the possibility of pipeline hazards and handle them accordingly. 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 for 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 some RISC instruction sets, the register operands have a fixed location in the instruction word 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 read a third operand from the register that is 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 feature is meant to simplify architecture implementations of the ARM ISA. Typically in pipelines, instructions take 2 cycles (fetch and decode) before they enter the execute stage. Thus, for single-threaded pipelines, the program counter has likely been incremented by 8. By using $instruction\_pc + 8$ as the operand value, the hardware implementation can directly use the PC 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 only incremented once every round robin cycle. Since $instruction\_pc + 8$ can be used as a data operand in the execute stage, the *PC Adder* in placed in the decode stage.

The *execute stage* contains the execution units and the multiplexers that select the correct operands to be fed into 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* performs 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 values are set in the bit vector. The memory addresses of each memory operation are 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 and still remove pipeline hazards. This improves the latency of the 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 be 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 result from the *ALU* is used to update the PC.

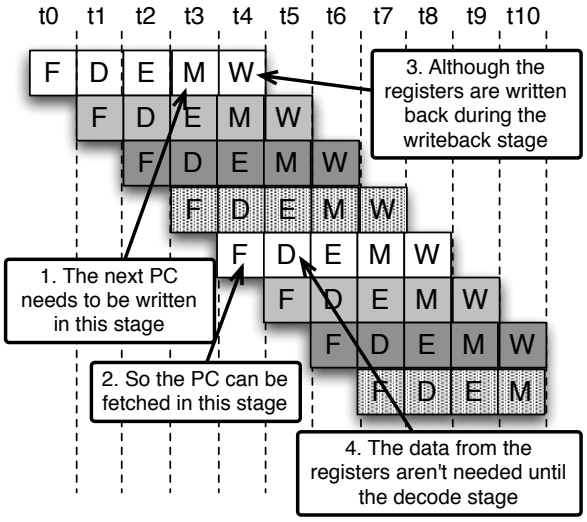


Figure 3.2: Four thread execution in PTARM

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 their 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 forwarding path is always used when a load instruction loads to R15. This does not stall the pipeline, and does not effect the timing of any following instructions. We describe the implementation 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. The memory map 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 are 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 set up 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 are reserved for user-registered exception handlers; these entries can be modified programmatically. For example, a location is reserved to store the address of a user registered *timer_expired* exception handler.

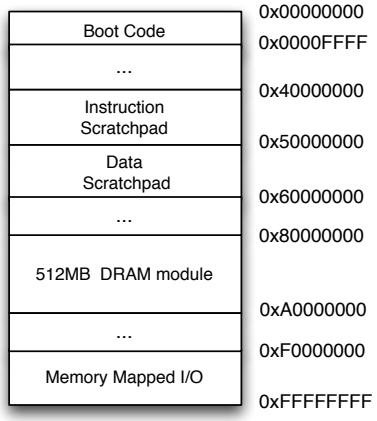


Figure 3.3: Memory Layout of PTARM

3.2.2 Scratchpads

Scratchpads replace caches as the fast-access memory in our memory hierarchy. The partition of instruction and data scratchpads between threads can be configured with 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 [110] of scratchpads and communication between hardware threads. This opens up opportunities to optimize system performance, instead of just individual thread performance. Hybrid schemes can also be used that privatize a hardware thread for security, and allow other threads to collaborate.

3.2.3 DRAM

The PTARM interfaces with a $512MB$ DDR2 $667MHZ$ DRAM memory module (Hynix HYMP564S64CP6-Y5). All accesses to the DRAM go 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

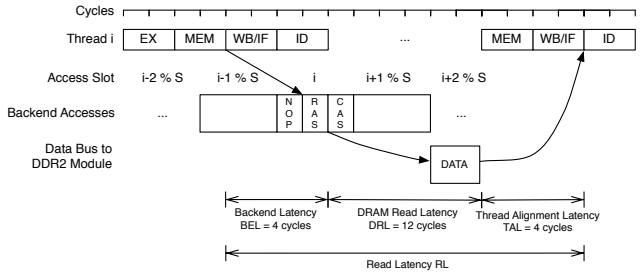


Figure 3.4: Example load by thread i in the thread-interleaved pipeline.

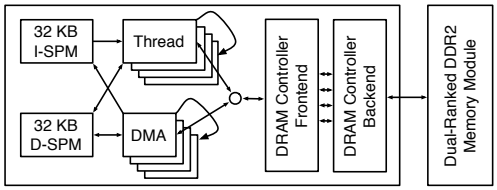


Figure 3.5: Integration of PTARM core with DMA units, PRET memory controller and dual-ranked DIMM [96].

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. The frontend of the DRAM controller also 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 the 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 that address 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 the DRAM is accessed through load (read) and store (write) instructions, the memory requests are issued directly from the memory stage of pipeline. Each 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 [96] we derive the access latencies from the alignment and show that even though memory access latencies can dependent on the alignment of the pipeline and the backend, they still exhibit only a small range of execution time. They are either 3 or 4 thread cycles. This is because the thread-interleaved pipeline hides the full memory latency with the interleaving of threads. We also leverage the 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 until after the load or store. In this case, the pushed back refreshes become invisible. Because the pipeline only reads back the data in the memory stage of the next instruction, it is not able to use successive access slots of the backend. Even if the the data is ready from the DRAM and put in the response buffer, it still needs to wait for the pipeline to reach the correct stage. Thus, the refreshes can be hidden in the successive unused access slot, and the frontend does not observe the refresh latency.

Whenever a DMA transfer is initiated, the DMA unit uses the thread's request buffer slot to service the DMA request to or from the scratchpad. Thus, while a DMA transfer is initiated, the

thread gives up access of the DRAM to the DMA unit. During this time, the thread can continue to execute and access the scratchpad regions that are not being serviced by the DMA request. This is possible because scratchpads are dual-ported, allowing a DMA unit to access the scratchpads simultaneously with 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 regions 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 the accesses from the pipeline, they suffer no alignment losses. When a refresh conflicts 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 in between DMA transfers are predictable, so the latency effects of the refresh can be easily analyzed, which we do in [96].

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 the DRAM. A bigger store buffer can hide the latencies of more successive stores at the expense of 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 input and output devices. 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 them. 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 [126]. 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

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

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 to the ARM ISA with our timing extensions. It is triggered when a user registered timestamp with *exception_on_expire* expires.

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. Thus, an exception that triggers on one thread must not effect the execution of other threads in the pipeline. In PTARM, any exception that occurs during instruction execution propagates down the pipeline with the instruction. The exception is checked and handled before modifying any pipeline states, such as the PC, CPSR, register, or 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 stored 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 is speculatively executed. This limits the timing effects 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.

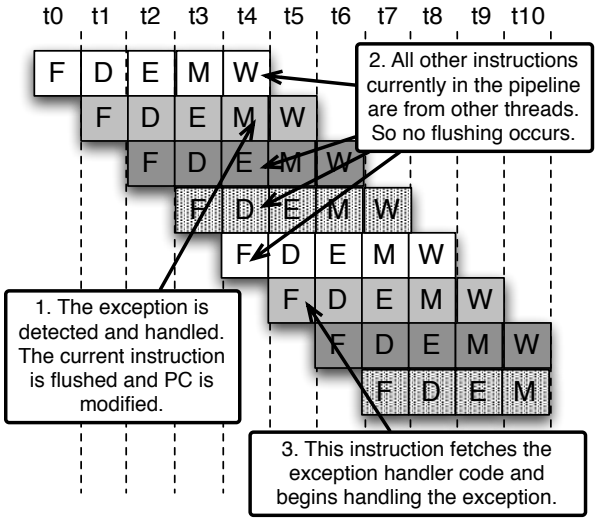


Figure 3.6: Handling Exceptions in PTARM

For longer latency instructions that modify the program state, exceptions can cause an inconsistent view of the program state. For example, a *timer_expired* exception could occur in the middle of a memory instruction to the DRAM. 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 already writing data to the DRAM. In this case, the programmer must disable the *timer_expired* exception 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 elapsed for the DRAM controller to finish servicing any potential memory requests. In PTARM, we ensure that the instructions executed before the worst-case memory latency in any exception handler 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 do not access the DRAM. We statically compile the data stack onto the data scratchpad, so any stack manipulations that occur also avoid 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 the details on each instruction type implementation 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.

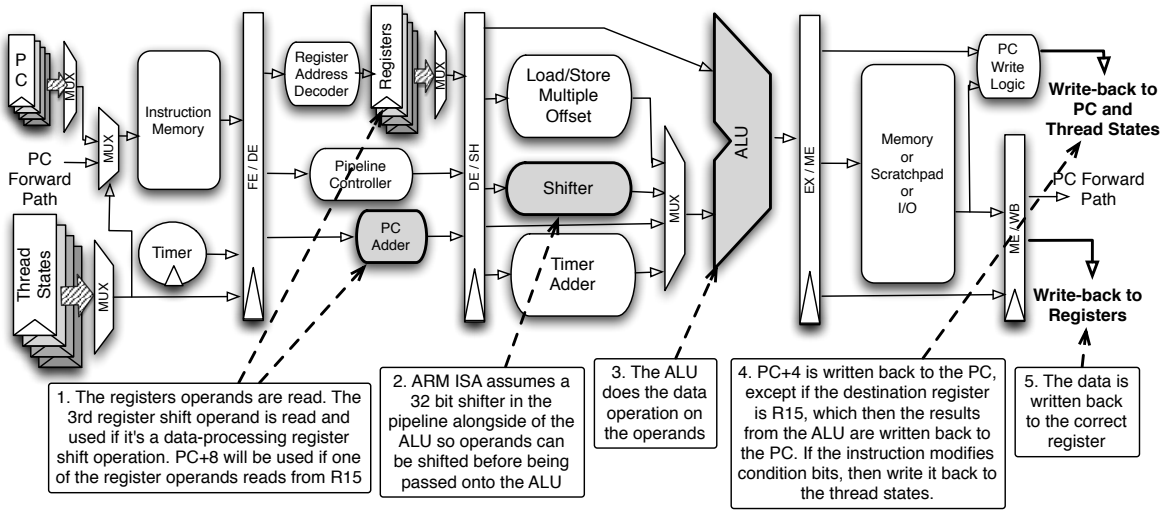


Figure 3.7: Data Processing Instruction Execution in the PTARM Pipeline

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, 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 they 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 zero (Z), carry (C), negative (N) and overflow (V) are used to predicate execution for ARM instructions. The high four bits of each instruction form 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 whether a conditional branch is taken or not. Figure 3.8 shows how branch instructions are executed in our thread-interleaved pipeline.

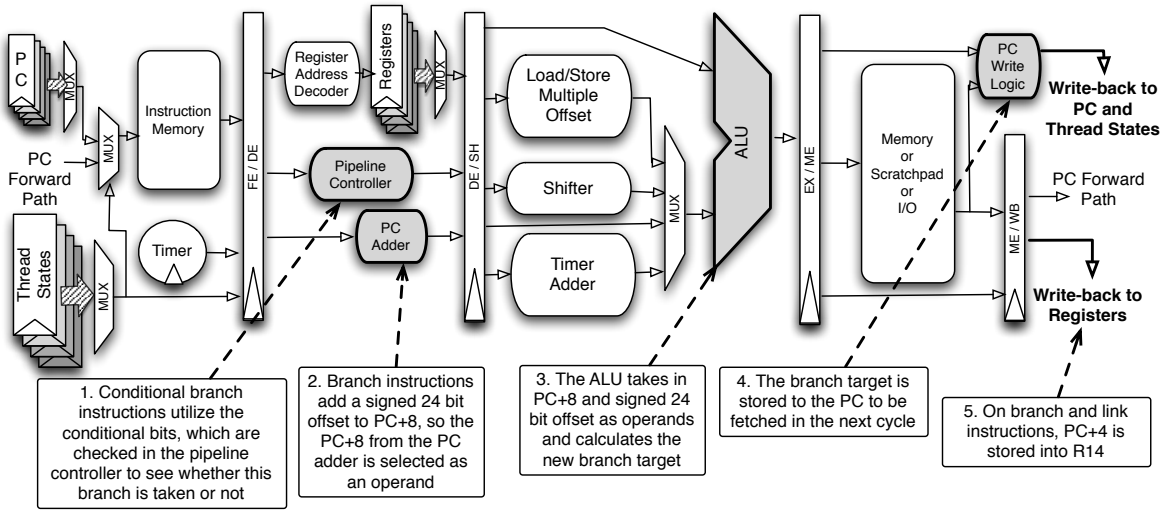


Figure 3.8: Branch Instruction Execution in the PTARM Pipeline

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. Branch and link (bl) instructions save the next address as a return address, so PC+4 is propagated down the pipeline and written back to the link register (R14).

All branch instructions, whether conditionally taken or not, take only one thread cycle to execute. But more importantly, the next instruction in the thread that executes 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 predicting the branch target address for 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 that loads the branch target address from memory. Although this slightly complicates our pipeline design,

we show that it does not affect the timing predictability and execution of the instruction nor subsequent instructions after the triggered branch. Currently load/store halfword and doubleword are not implemented in PTARM, but 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 utilize 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 support word or byte operations. Figure 3.9 describes how the load/store register is implemented in the pipeline.

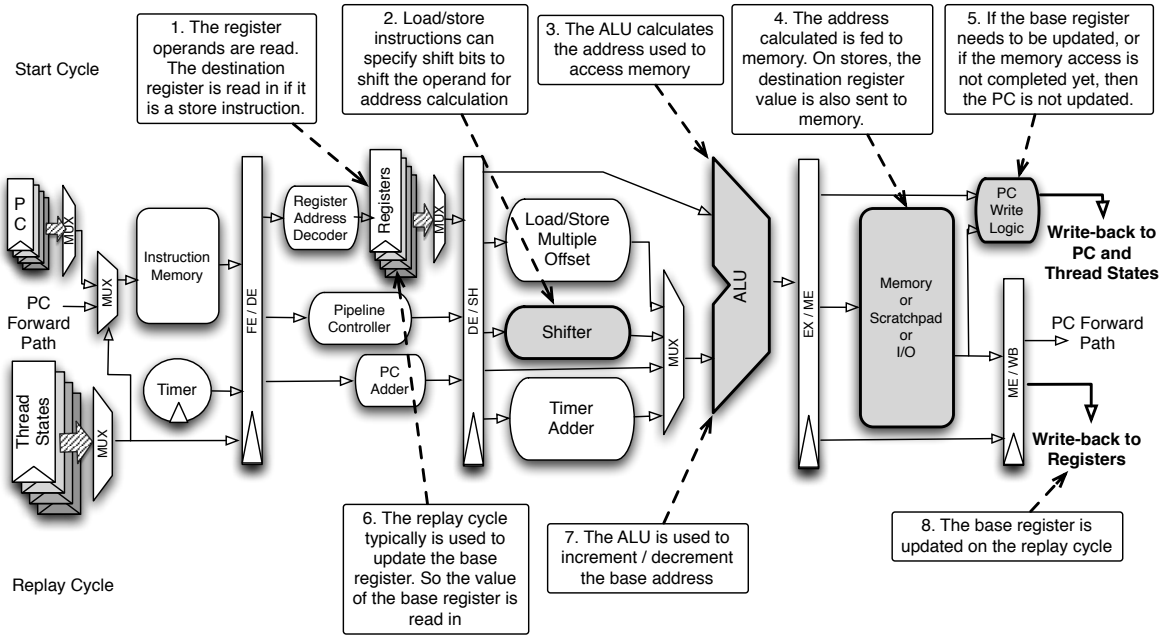


Figure 3.9: Load/Store Instruction Execution in the PTARM Pipeline

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 a 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 the 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 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 update 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 updating 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. 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.

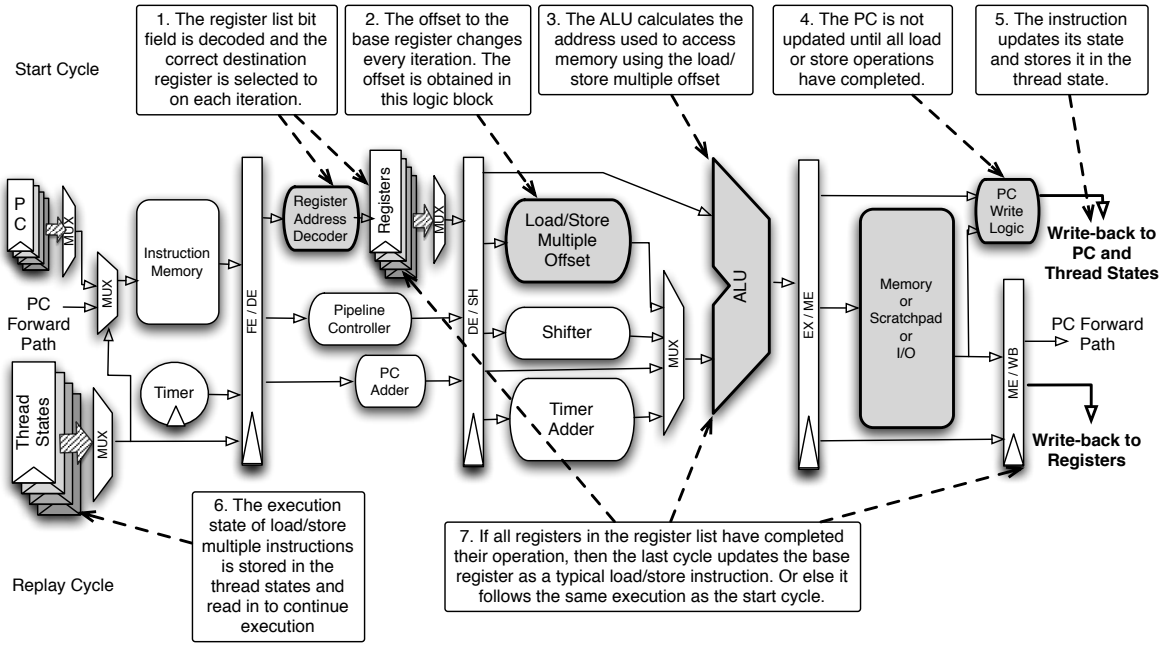


Figure 3.10: Load/Store Multiple Instruction Execution in the PTARM Pipeline

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 to 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 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, then each register load/store will take multiple cycles. Load/store multiple instructions can also update the base register after all the register operations complete. 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 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. 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 to use the forwarded PC on the next instruction fetch, instead of the one stored in next PC. We show 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 discusses the timing implications of data-forwarding logic in the pipeline. Although it seems the selection of PC is dynamic, 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, because 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 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

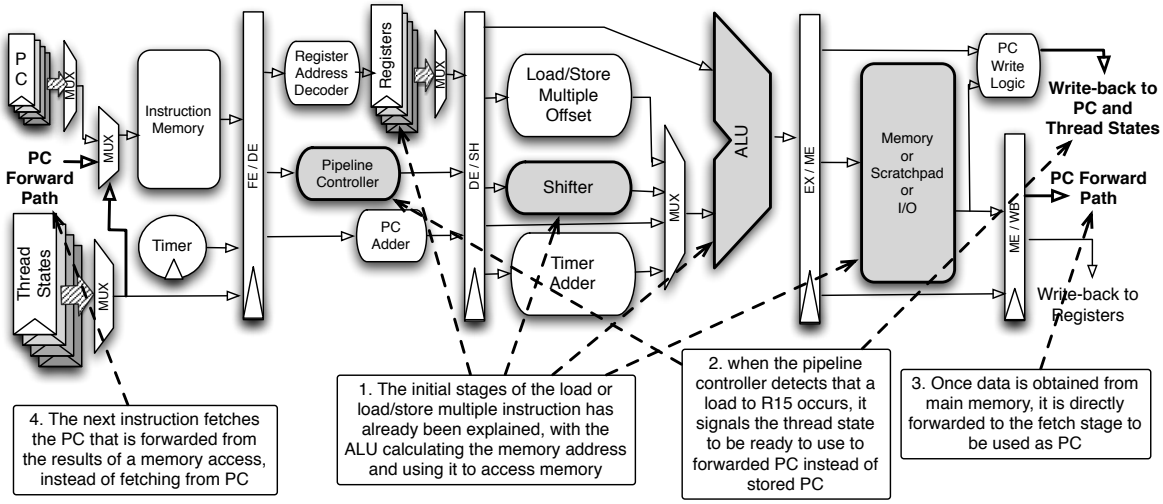


Figure 3.11: Load to R15 Instruction Execution in the PTARM Pipeline

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 gives the instruction extensions to the ARM ISA that bring timing semantics to 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

Type	Opcode	Functionality

get_time	8	timestamp = *current_time*;
		crd = high32(timestamp);
		crd+1 = low64(timestamp);

delay_until	4	deadline = (crm << 32) + crn;
		while (*current_time* < *deadline*)
		  stall_thread();

exception_on_expired	2	offset = (crm << 32) + crn; register_exception(offset);
deactivate_exception	3	deactivate_exception();

Table 3.2: List of assembly deadline instructions

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 and predictability of the 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 to 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 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. This privileged instruction simply modifies the timer increment, thus we omit the implementation details due to its trivial implementation.

The timestamp associated with each instruction execution is latched during the fetch stage of the pipeline. In other 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 operations 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 is not affected by the timing instructions from other threads. With 4 hardware threads interleaving 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 implements 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, *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.

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 form the 64 bit timestamp checked against the platform clock every thread cycle. As described in section 2.3, the *delay_until* instruction can be

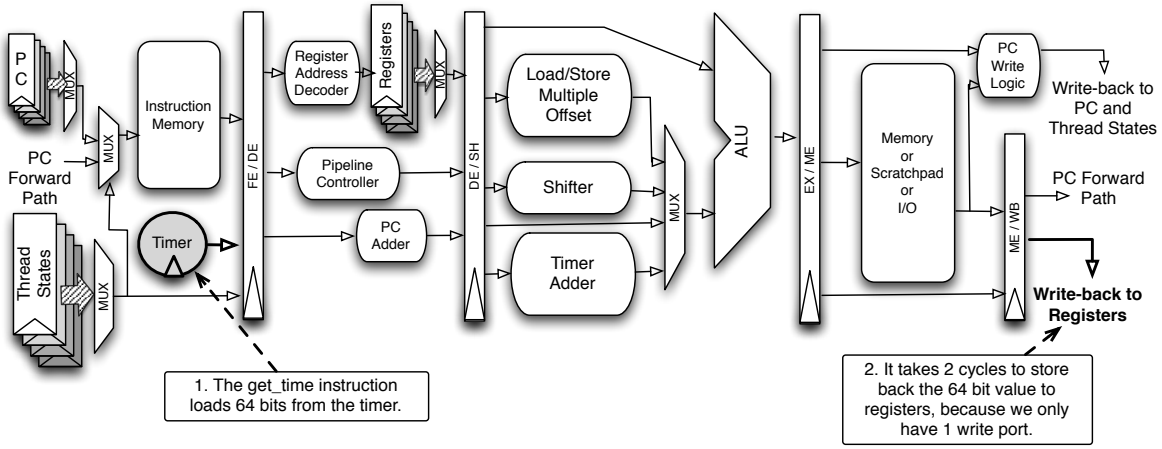


Figure 3.12: Get_Time Instruction Execution in the PTARM Pipeline

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.

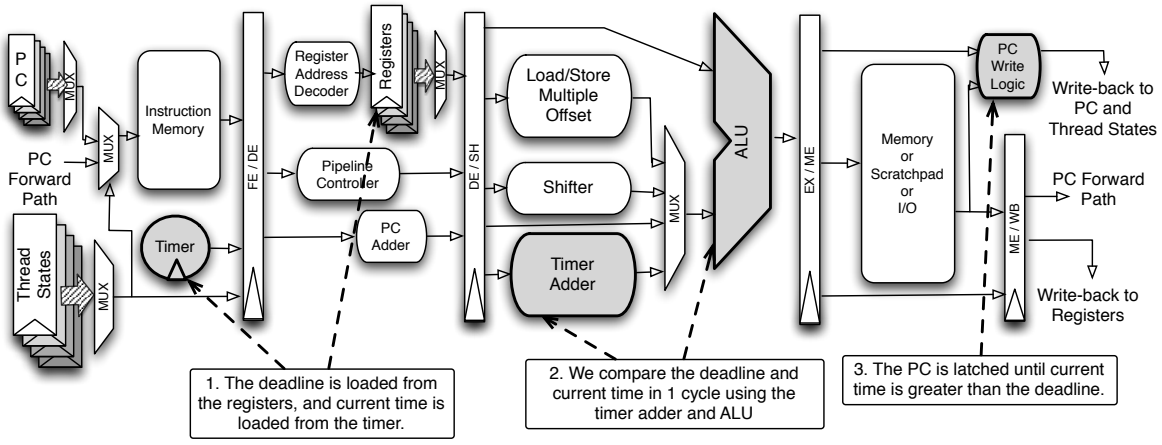


Figure 3.13: Delay_Until Instruction Execution in the PTARM Pipeline

The *delay_until* instruction highlights the reason *timer adder* is added into the pipeline. During the execution of *delay_until*, the platform clock value is compared every thread cycle to 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 ensures that no additional thread cycles elapse after the input timestamp is reached. To delay program execution, the PC is only updated when the clock value is greater then 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 the setting of timer interrupts on embedded platforms, which is 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 than 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 not active, then *deactivate_exception* will do nothing. The implementation of the *timer* unit is shown in figure 3.14.

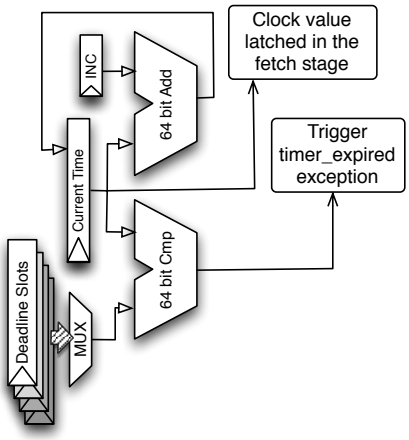


Figure 3.14: Implementation of Timer Unit

Exception_on_expire and *deactivate_exception* instructions are thread specific; each thread has its own dedicated deadline slot. The handling of *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 executing in. The timing details and jitter introduced with this mechanism are detailed in section 3.6.2.

Each thread currently can only check for one timestamp in hardware. To create the effects of multiple timestamps 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. 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. We synthesize the PTARM on the Xilinx ML505 [134] evaluation board, which includes the Virtex-5 XC5VLX110T FPGA [132] and several I/O interfaces on the board. PTARM connects to the on board LEDS, RS232 connector, DVI transmitter device 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. We also include the Xilinx Integrated Logic Analyzer (ILA) to be used for debugging the pipeline and memory controller. All VHDL source code, software code samples, and instruction manual can be downloaded from http://chess.eecs.berkeley.edu/pret. Figure 3.15 shows the high level block diagram of the PTARM soft core.

PTARM communicates with the current I/O devices through memory mapped control registers. Each control register can be accessed within a single cycle, so no contention arises on the I/O bus. The LEDs are memory mapped and can be toggled by setting and clearing bits. PTARM interfaces to the UART through the UART gateway, which queues read and write requests from the core and relays it to the UART. 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 are done through blocking procedure calls. The UART runs at a baud rate of 115200, and sends and receives bytes.

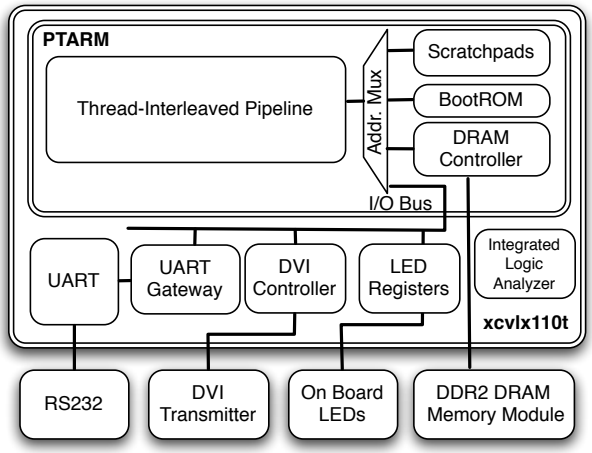


Figure 3.15: PTARM Block Level View

The DVI controller interfaces with the Chrontel CH7301C DVI transmitter device [5] on the evaluation board. We initialize the DVI transmitter to RGB bypass mode to manually supply the sync signals to the DVI output. A software DVI controller similar to the one presented in [49] has been implemented, where the VGA sync signals are managed 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 are used to render pixels and draw to the screen buffer. Because hardware threads are temporally isolated, the timing of hardware sync signals is not affected by the operations on other hardware threads.

We target a $100MHz$ clock rate for the PTARM pipeline and a $200MHz$ clock rate for memory controller to interface with the DDR2 DRAM. We compare the resource consumption to the Xlinx MicroBlaze [130] soft processor platform generated from the Xilinx Embedded Development Kit (EDK) [131]. We also target the MicroBlaze at a $100Mhz$ clock rate, and choose to optimize the area over performance. We configure the MicroBlaze platform to include a DDR2 DRAM controller, a UART controller and a generated LED controller. The MicroBlaze also includes a

local memory block (LMB) and instruction and data caches. For fair comparison, we configure the caches and local memory block to have similar sizes to the scratchpads and boot memory on PTARM.

We use the Xilinx Virtex-5 XC5VLX110T FPGA to implement both PTARM and MicroBlaze. Each Virtex-5 logic slice contains four 6-input look up tables (LUT6s), four flip flops (FFs), muxes, and carry chains. The FPGA also includes block RAMs (BRAMS), which are dedicated memory blocks, and DSP slices, which are special logic slices for DSP or other computational functions. The BRAMs are used to implement the scratchpads and register file for PTARM. The physically duplicate copies of the register file for each hardware thread warrant the use of BRAMs to save on logic slices. We use the DSP slices to implement the most of the timer increment and comparison functions.

	PTARM									
	Pipeline				**DRAM Interface**		**Peripherals**			
	Shifter	Timer	ALU	**Total**	Controller	**Total**	UART	DVI	ILA	**Total**
LUTs	288	35	242	1615	286	1551	63	76	806	4134
FFs	0	84	0	1042	607	2181	49	84	837	4439
DSP Slices	0	4	1	6	0	0	0	0	0	6
	SPMs	Boot	Registers	**Total**	Controller	**Total**	UART	DVI	ILA	**Total**
BRAMs	6	1	3	10	0	2	0	0	9	21

	Microblaze									
	Pipeline				**DRAM Interface**		**Peripherals**			
	Shifter	Timer	ALU	**Total**	-	**Total**	UART	LEDs	-	**Total**
LUTs	96	-	35	1831	-	2175	130	122	-	4402
FFs	36	-	0	1433	-	3049	124	194	-	5032
DSP Slices	-	-	-	3	-	0	0	-	-	3
	Caches	Local	Registers	**Total**	-	**Total**	UART	LEDs	-	**Total**
BRAMs	6	1	-	7	-	13	0	0	-	20

Table 3.3: PTARM and Microblaze Resource Usage on the Xilinx Virtex5 ML505 Evaluation Board

Table 3.3 shows that the resource consumption of PTARM is similar to the area optimized MicroBlaze. PTARM uses slightly less LUTs and FFs for the pipeline, as the data and control hazard logic are stripped out, and the cost of the extra copies of the register file is absorbed by the BRAM implementation. The timer added to extend the ISA with timing semantics uses mostly DSP slices to implement the platform clock and comparator for timing exceptions. The BRAMs used for scratchpads vs caches were the same because we configured the sizes to be similar, but the scratchpads also saved on the logic slices used to implement the hardware replacement policies of caches. Thus, even though the MicroBlaze implements a more optimized ALU and shifter compared to PTARM, the PTARM pipeline still uses fewer resources. The critical path of our pipeline is at the execute stage, which includs a serial connection of the 32 bit barrel shifter and the ALU. To further improve the clock frequency, we can split up this stage into 2 stages, one for the shifter and one for the ALU, at the cost of one additional hardware thread. We show this in chapter 4 for an engine fuel rail simulation application, in which we clock a six thread six stage thread-interleaved pipeline at $150MHz$.

The PTARM DRAM Interface is based on the Xilinx core generated [129] DRAM In-

terface, which is also what is used in the MicroBlaze architecture. We replace the queuing and reordering logic in the generated DRAM controller with our own front and back end implementation of bank privatization. The slice consumption is shown in the table labeled "controller" under the DRAM interface. It shows that our predictable DRAM interface uses fewer logic slices and BRAMs than the original DRAM controller.

Although these results may vary slightly depending on the synthesis toolchains, settings, and versions used, it gives us a general estimate of the resources consumed by our predictable architecture. The resource comparisons confirm our conjecture that a predictable thread-interleaved pipeline, scratchpads and memory controller can lead to similar or less resources compared to conventional architectures that use hardware techniques to optimize average case performance.

3.5.2 PTARM Simulator

Along with the VHDL soft core of our architecture, we also provide a cycle accurate C++ simulator, which can also be downloaded from http://chess.eecs.berkeley.edu/pret. The simulator faithfully models the five stage thread-interleaved pipeline and its interaction with the memory hierarchy, including scratchpads and the predictable DRAM controller. The simulator is mainly used for software experimentation and architecture exploration. The DMA units described in section 3.2.3 are currently implemented only in the simulator, as we are still exploring the architectural design to make DMA transfers from scratchpad to the DRAM predictable. The timing instructions are also implemented in the simulator to allow for software experimentation of the ISA with timing semantics.

To evaluate the performance of our architecture, we used the Malardalen WCET benchmarks [40] and compare our simulator against the SimIT-ARM [92] cycle-accurate simulator. The SimIT-ARM simulator simulates a StrongARM 1100 [48], which contains a five stage pipeline, branch delay slots without branch prediction, a 16kb instruction cache and a 8kb data cache. We configure our PTARM simulator to use similar sizes for the instruction and data scratchpad. The StrongARM1100 is implemented with $0.35um$ process technology, and can be clocked from 133 MHz to up to 220 MHz, Although we currently clock PTARM at 100MHz, it is implemented on an FPGA, and not silicon. Thus, we use clock cycles as our unit of measurement in our experiments. Both architectures implement the ARMv4 ISA, so we used similar ARM cross-compilers to compile the benchmarks for both architectures. In this way, the compiler or ISA played no effect on the performance differences. Because the Malardalen benchmarks are single threaded, we set up our experiments to run the same benchmark on all four threads of the PTARM architecture, and four times sequentially on the single threaded SimIt-ARM simulator. This way, the total number of instructions executed on both architectures are roughly the same, and the setup mimics an embarrassingly parallel application.

Most of the benchmarks we choose fit entirely within the scratchpad/cache of the architectures. This is intentional, as a full system evaluation of scratchpads vs caches involves several factors including the scratchpad allocation scheme, and is beyond the scope of the thesis. We thus mainly measure the effects of the thread-interleaved pipeline compared to the StrongARM1100's single threaded five stage pipeline. For the benchmarks that do not fit entirely within the scratchpad for PTARM, we profile and statically compile the most frequently used memory locations onto the scratchpad. Because the StrongARM1100 uses instruction and data caches, it suffers from a cache cold start, so the initial run of the benchmarks suffers more cache misses to load the instructions

and data onto the caches. “SA1100 cold” denotes the measurement of four runs including the cold start. To mitigate the performance effects from the cold start, we warm up the cache by first running the benchmark once, then measuring four sequential runs of the benchmark on the StrongARM100. This is labeled as “SA1100 warm” in the figure. To further remove the effects of caches from the StrongARM architecture, we adjust the memory access latency to 0 cycles, to appear as if every memory access were to the cache. This is labeled as “SA1100 allcache”. We obtained the cycle counts for both architectures, and compare the instruction throughput, shown in figure 3.16, and overall latency, shown in figure 3.17, for several benchmarks.

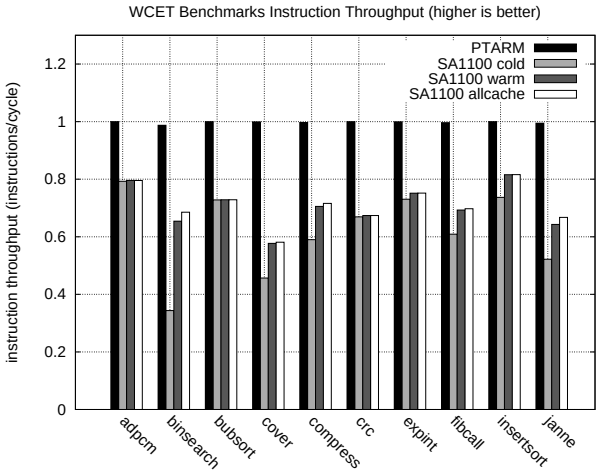


Figure 3.16: Malardalen benchmarks throughput

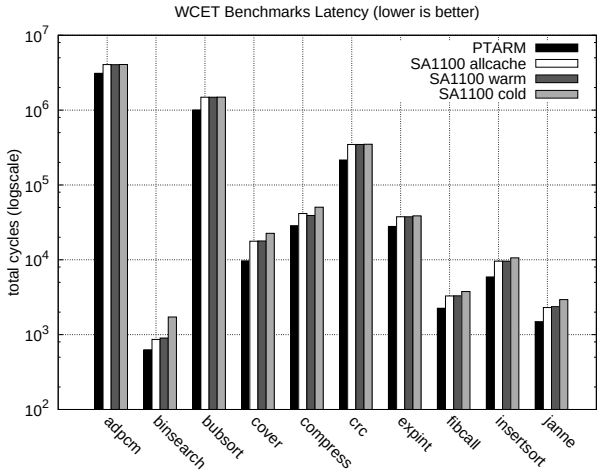


Figure 3.17: Malardalen benchmarks latency

Several observations can be made from these measurements. First, we observe from figure 3.16 that PTARM achieves almost an one instruction/cycle throughput for all benchmarks. The thread-interleaved pipeline removes the control and data hazards from within the pipeline. Thus, the pipeline almost never stalls. On the contrary, with the single threaded StrongARM1100, the effects of pipeline hazards reduce the throughput of instructions, as the pipeline needs to stall for control and data hazards that can arise. With the higher instruction throughput, PTARM observes a smaller latency in terms of clock cycles for all benchmarks executed, as shown in figure 3.17.

Second, we observe the effects of caches on the execution time variance by comparing the throughput and latency of “SA1100 cold” and “SA1100 warm.” The greatest execution time variance can be observed from *binsearch*, which performs a binary search on an array of 15 elements. In this particular benchmark, the throughput and latency difference is a factor of two between the warmed up cache and the cold started cache. By comparing the throughput of “SA1100 warm” and “SA1100 all cache,” we can observe that most benchmarks fit entirely in the cache, as the throughputs are roughly the same. Because the memory hierarchy is hidden by caches, even though most benchmarks fit entirely in the cache, we cannot avoid the cold start because the programmer has no control over the cache. With an exposed memory hierarchy in PTARM, we statically compile the benchmarks on the scratchpad, and are able to benefit from the smaller code size without suffering the effects of cold starts. We also maintain a deterministic execution time.

The higher instruction throughput achieved by interleaving hardware threads in the pipeline comes from trading off single thread latency. The thread-interleaved pipeline time-shares the pipeline

resources between the hardware threads, so the latency of a single thread is slower compared to a single threaded pipeline. But for applications with enough parallelism to fully utilize the pipeline, the higher instruction throughput gives better overall performance. We show one such application in section 4.1 that models, in real time, an engine fuel rail to enable more precise fuel injection for combustion engines. A thread-interleaved pipeline also allows us to clock the pipeline at a higher frequency, because the data hazard handling logic can be stripped out of the pipeline, providing less logic within each pipeline stage. Thus, with a higher instruction throughput and higher clock speed, timing predictability and composability can be achieved without sacrificing performance. We also highlight the uncontrollable execution time variance of a hidden memory hierarchy using caches. We do not claim that scratchpads will always provide better average case performance, as a full performance comparison between scratchpads and caches is outside of the scope of this thesis. However, with an exposed memory hierarchy, we are able to control and remove the execution time variance by statically compiling instructions and data onto scratchpads, providing timing determinism for memory accesses. For the predictable DRAM controller, Reineke et. al [96] show that bank privatization of DRAMs not only achieves predictable DRAM access latencies, but also lowers worst case access latency, and improves throughput and average case memory latency under high contention. These results demonstrate that one does not need to forgo performance in order to achieve timing-predictability in architecture design.

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 execution time of the worst case path on the underlying architecture. A plethora of research has been done on the software analysis of program paths. Wilhelm et al. [125] present a survey of tools and techniques available for worst-case path enumeration, loop analysis, etc. However, the precision of the WCET analysis of those techniques ultimately depends on the underlying architecture implementation [43]. 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 3.4 summarizes the execution time each instruction takes in terms of *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.

Instruction	Latency	Instruction (*Addressing Mode*)	Memory Region Accessed	
			SPM/Boot	DRAM
Data Processing	1	Load Register (*offset*)	1	4^{ϕ}
Branch	1	Load Register (*pre/post-indexed*)	2	5^{ϕ}
Software Interrupt (SWI)	1	Store Register (*all*)	1	2^{δ}
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, making it 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.

†: This is the minimum execution time of *delay_until*. The actual execution time varies depending on the input timestamp.

Table 3.4: Timing properties of PTARM instructions (in thread cycles)

3.6.1 Memory instructions

Data-processing and branch instructions have straightforward execution times. The execution time of branches is 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 3.4. 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 [125]. 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 and more accurate timing 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 whether the access hits or misses 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 access 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 instruction to see whether it is a memory accessing instruction to the DRAM. Since only one instruction needs to be checked, it only slightly complicates the timing analysis. If it is not possible, then the full 2 cycle 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 they issue 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 like the load/store register 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 is static. However, the timing instructions can impact the execution time of 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 changes. 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 timestamps that represent 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 represents 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 stored 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* delays programs execution until the current *time of execution* is greater than or equal to the input timestamp value. When *delay_until* completes its execution, the next instruction will observe a platform time of *at least* $40ns$ greater than the input timestamp passed to *delay_until*. This effect is illustrated in figure 3.18.

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* contains 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 less than or equal to 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 greater than $t + 160$. This is the top case shown in the figure. The instruction after *delay_until* executes at time $t + 200$, which accounts for the 1

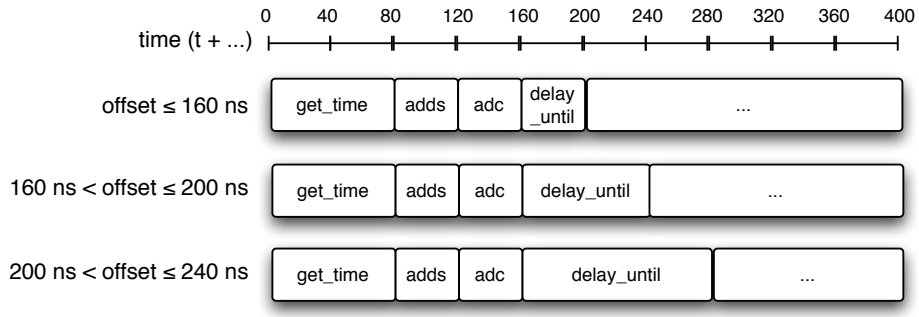


Figure 3.18: Timing details of get_time and delay_until

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 3.18. *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 3.19. If the thread has a deadline of

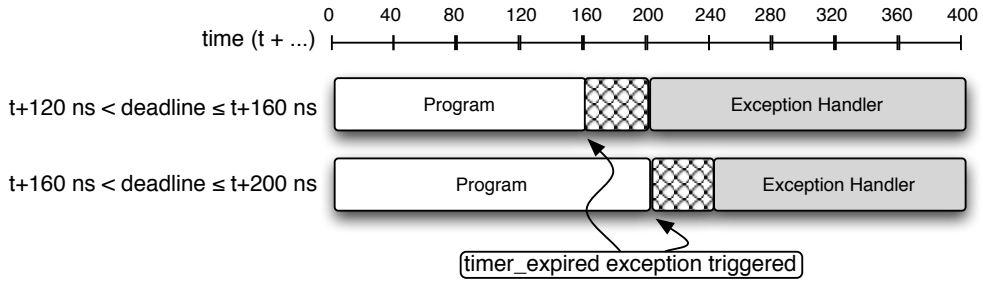


Figure 3.19: Timing details of the *timer_expired* exception triggering

$t + 161ns$, 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

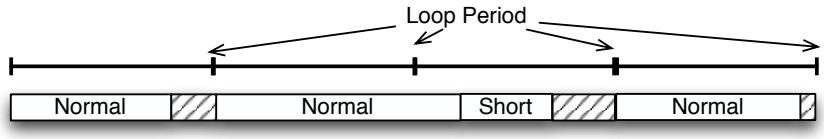


Figure 3.20: Execution of the self compensating timed loop

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 r5, r3, r5             ; Check whether previous iteration deadline is missed
8   sbc  r4, r2, r4             ;
9
10  blmi task_short             ; execute shorter task if previous deadline miss
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 iteration
15  cdp p13, 4, c2, c2, c3, 0   ; delay_until
16
17  b loop
```

execute a shorter version of the task in attempt to make up for the lost time, as shown in figure 3.20.

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 3.5, where we show a sample execution trace of four iterations in this timed loop. Figure 3.20 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 3.5, execution starts at time t. As mentioned before, each thread cycle is $40ns$, which is reflected 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 3.4. All instructions are statically compiled onto 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).

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 r5, r3, r5*	compare deadline (t) and current (t)
t+280 ns	n+7	*sbc r4, 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 r5, r3, r5*	compare deadline (t+4000) and current (t+4000)
t+4280 ns	n+107	*sbc r4, 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 r5, r3, r5*	compare deadline (t+8000) and current (t+8040)
t+8320 ns	n+208	*sbc r4, 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 r5, r3, r5*	compare deadline (t+12000) and current (t+12000)
t+12280 ns	n+307	*sbc r4, r2, r4*	result is 0, clear cc[“n”]

Table 3.5: Instruction execution trace of the self compensating timed loop
(TC = thread cycles)

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 the *deadline_timestamp*. In general, the *offset* that needs to be accounted 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 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 determine which task to execute. From table 3.5 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* executes for 89 thread cycles, exactly one thread cycle over its timing requirement. As a result, the *delay_until* of the second loop iteration does 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 includes 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 3.21. We assume that the setup code remains 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 exe-

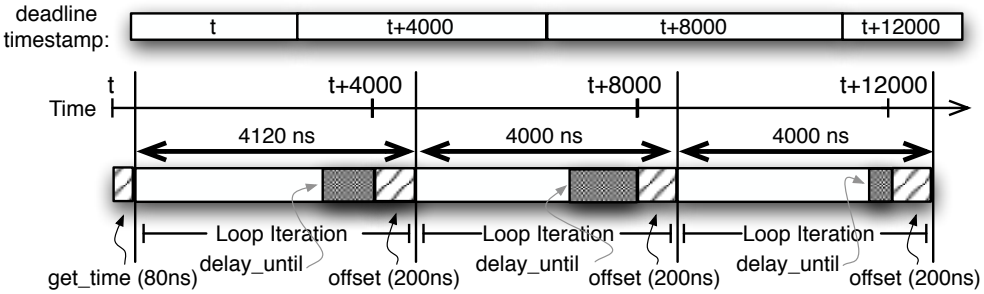


Figure 3.21: Jitter caused by initial timed loop setup

cution 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 r4, r3, r5             ; Check if previous iteration deadline is missed
13  sbc  r5, 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 )
```

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 3.22, where the first

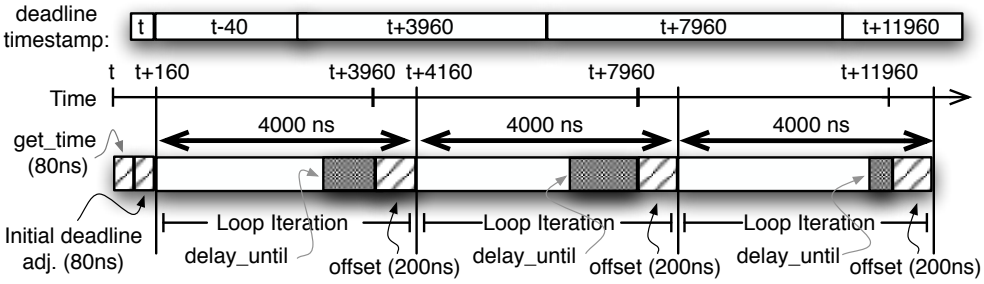


Figure 3.22: Adjusted timed loop setup

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 occurs, 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 where 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 3.6. 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 it 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* exception (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, statically compiled on 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 ac-

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 3.6: Exception_on_expire sample code timing details

cessed 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 suffers 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 *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 3.6.

In this chapter, we present the PTARM, an implementation of a precision timed machine utilizing the principles described in chapter 2. Along with a full description of the architecture, we show that each instruction in the PTARM exhibits deterministic execution times, and the hardware threads in the PTARM are temporally isolated. Our benchmarks show that when the architecture is fully utilized, the PTARM achieves higher instruction throughput compared to the single threaded ARMv4 StrongARM1100 architecture, proving competitive performance with timing predictability. We demonstrate the predictable execution time behavior and simplified timing analysis through several examples that use the timing extensions to the ISA. These also demonstrate the use and possibilities of the timing extended ISA.

Chapter 4

Applications

In this chapter we 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 an improved engine efficiently when injecting fuel. The application makes use of the lightweight hardware threads in our thread-interleaved pipeline to implement a massively parallel simulator with hundreds of computational nodes communicating to their neighbors. The timing predictable architecture allows us to statically analyze the execution time to ensure that each computational node can meet the timing constraints imposed by the application. A time based communication scheme is used to reduce communication overhead. The communication synchronization is enforced in software with timing instructions. This minimizes overhead and ensures that the communication of all nodes are in sync. We present the synthesis results to show that by using the PRET architecture we can successfully simulate a common fuel rail configuration of up to 234 nodes on a Xilinx Virtex-6 FPGA, and enable real-time engine fuel rail simulation. This work is joint research in collaboration with Guoqiang Gerald Wang and Hugo Andrade, from National Instruments Corp., and Matthew Viele, from Drivven Inc. The results are also published in [68].

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 the variations in execution time of crypto-algorithms to deduce the encryption keys. The root cause of these time-exploiting attacks is the uncontrollable run-time variations that are 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 over execution time, we remove the vulnerability that is exploited to initiate the attack, and remove architecture deficiencies that can lead to more timing-attacks. We demonstrate this by running the RSA and DSA encryption algorithms on PRET, and successfully illustrate 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 valve is used to control the amount of fuel injected, which depends on the pressure and fuel rate within the fuel rails delivering the fuel. Several pilot injections are

injected ahead of the main injection to mitigate the injection 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 that the correct amount of fuel is injected [19]. Currently, fuel rails are modeled and developed with 1D-CFD solvers like GT-Fuel. They use an ad-hoc model of fuel pressure for injection events [127]. 1D-CFD models are commonly used when simulating the transient operation of internal combustion engines [106]. Here, we present an implementation for a 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 described as a network of pipes. The system is built up from different types of pipe segments, each modeling 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 their neighboring pipe segments. The time step is determined by the speed of information flow, 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 Δx is the discretization length. For stability, the courant number needs to be less than 1, and a number below 0.8 is recommended [38]. For example, if a fluid has a wave speed a of 1 cm per microsecond and a discretization length Δx of 1 cm, then we require a time step Δt of less than one microsecond. This discretization length of a pipe network is dominated by its smallest sub-volume. A discretization length of 1 cm and a speed of sound (wave speed) of 1500 m/s [111] are commonly used for diesel fuel systems. The real-time requirements of this application thus require adequate performance so that the slowest node can complete in Δt. 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 [136], such as in image processing applications.

In order to evaluate our system of pipes, we define a few types of computing nodes that correspond to different pipe elements. These are shown in table 4.1 along with their derived pressure

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	$\frac{\frac{C_{P_1}}{B_1} + \frac{C_{M_2}}{B_2} + \frac{C_{M_3}}{B_3}}{\sum \frac{1}{B}}$	$-\frac{P_I}{B_1} + \frac{C_{P_1}}{B_1}$ $-\frac{P_I}{B_2} + \frac{C_{M_2}}{B_2}$ $-\frac{P_I}{B_3} + \frac{C_{M_3}}{B_3}$

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

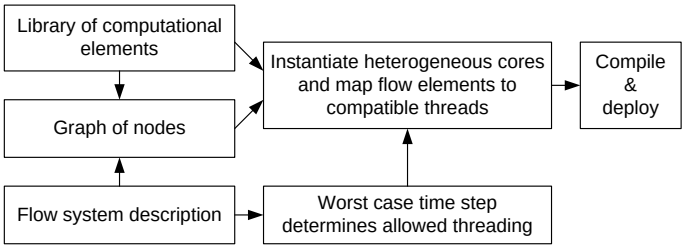


Figure 4.1: Design Flow

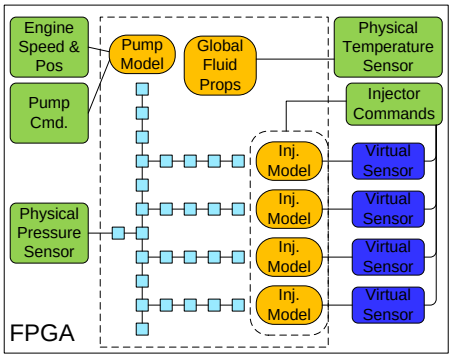


Figure 4.2: High Level System Diagram

and flow rate equations. From these pipe elements we can generate a network of pipes that represent 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 a 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 [116].

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

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

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 the pressure, ρ is the fluid density, V is the fluid velocity, f is the Darcy-Weisbach friction factor, D is the 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 τ, the fraction of the valve that 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 timing requirements.

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 determines 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, and its graphical representation is shown in Table 4.4. The pipe network starts with an imposed input flow (P1) element on the left that 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

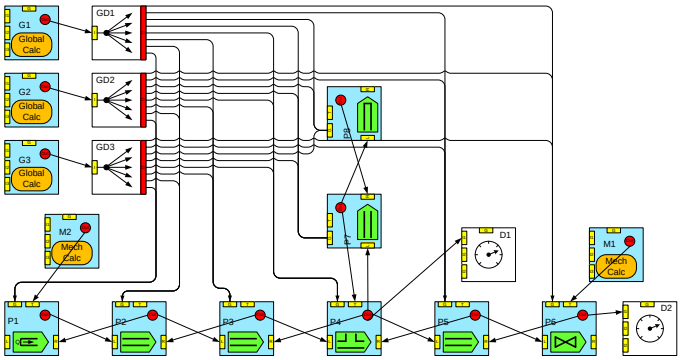


Figure 4.3: Detailed System Diagram

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

world (D1) through an 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 continuing down the original path exits the system through a valve (P6). *Mechanical calculation* elements compute the inputs to the valve, the defined flow, and the 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). These 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 serve 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 able to scale 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 shared local memories on an FPGA. Computational nodes are mapped to hardware threads on the PTARM cores. The PTARM cores used for this application are a slightly modified version of the one presented in chapter 3. In order to improve the throughput and clock frequency of our pipeline, we implement 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 interleaving through the pipeline. The memory footprint for each of the computational nodes range from roughly 100 to 1000 bytes. Thus, the scratchpad memories within a PTARM core are sufficient to hold all instruction and data for all threads; no external memory is required.

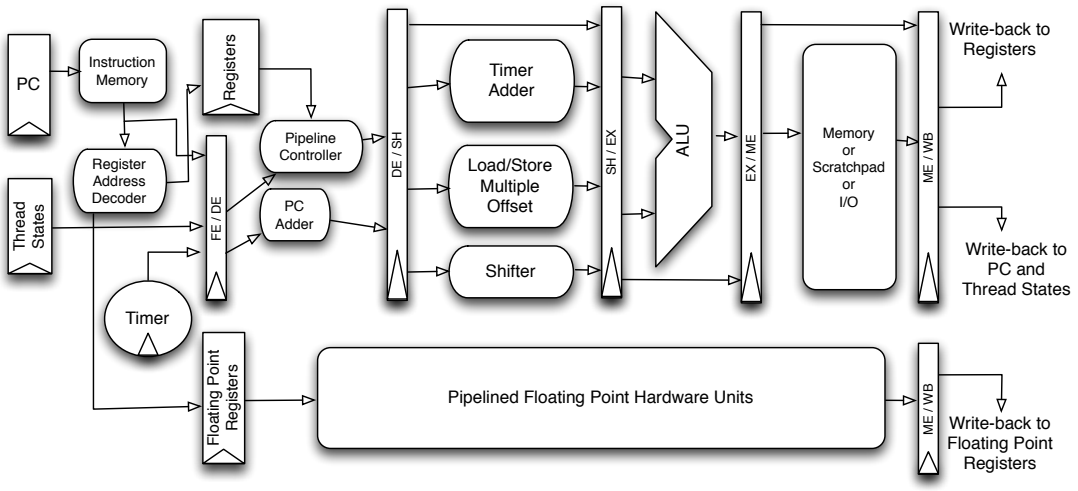


Figure 4.5: The PTARM 6 Stage Pipeline

The pipeline also contains hardware floating point units to support the application's need of floating point computations. The floating point units used are single-precision, and generated using the Xilinx Coregen tool [129]. They are pipelined to accept inputs every cycle to avoid 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 point 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. Even though the floating point divide and square root are the most resource intensive hardware units, there are usually only a few valve and "T" nodes in the overall system. The common fuel rail system we 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 increasing 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 the PTARM core, we can include the extra hardware units 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 hides 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 computed pressure and flow rate values 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.

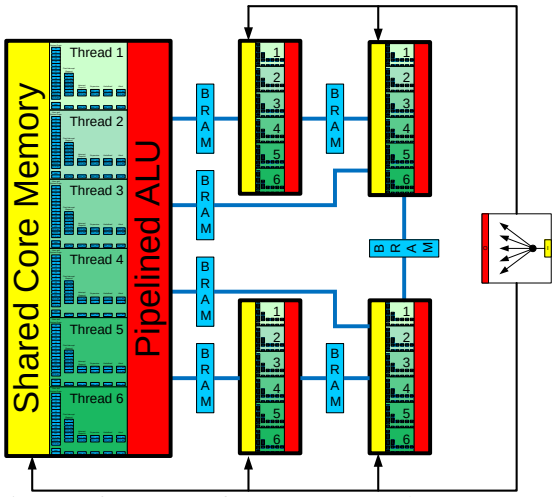


Figure 4.6: System of PRET Cores and Interconnects

Because nodes are mapped to the hardware threads on the PTARM cores, their neighboring nodes may be mapped to threads on the same core, or threads on a neighboring core. Nodes mapped to the same core (intra-core communication) communicate through the shared scratchpad memory within the core. Nodes mapped to different cores (inter-core communication) use privately shared Block RAMs (BRAMs) between cores to establish point-to-point communication channels. 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 in a single cycle, 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 FPGA designs, the limiting resource is typically logic slices, not the BRAMs. Each PTARM core only requires a small number of BRAMs for registers and scratchpads, so the BRAM utilization ratio is far less than the logic slice utilization. We present our synthesis results in section 4.1.3, which show that the number of cores synthesized is indeed limited by the logic slices, not the BRAMs.

When implementing the global distribution circuit, we observe 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-ported 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 the same values to all the broadcast memories. The broadcast memory for each core is shared amongst all threads in the 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, and any network protocol for the purpose of broadcasting. Figure 4.6 shows a block-level view of the hardware architecture.

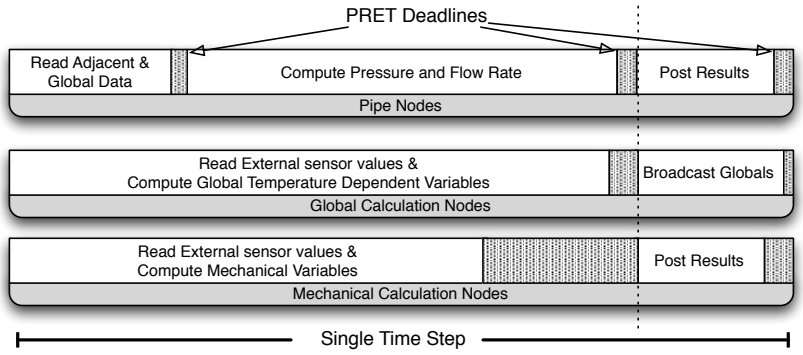


Figure 4.7: Execution of Nodes at Each Time Step

Software Architecture

We implement the equations shown in table 4.1 in C, and compile them with the GNU ARM cross compiler [3] to be 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 are 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 the pressure and flow rate values from neighboring nodes, and the temperature dependent variables from the global broadcasters. Once input values are read, the computations occur according to the specific fluid dynamics equations. In 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 avoid data races and ensure that all data is consistent and from the same time step. This communication model is very similar to Giotto [45], where tasks communicate explicitly through ports, and only at the end of the execution of 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 execution model.

In section 2.3 we introduce 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 the *delay_until* instruction. The semantics of the *delay_and_set* macro is similar to the deadline instruction introduced by Ip and Edwards [49]. It first enforces a previously specified timing constraint, then it sets a new timing constraint for the next code block. The *delay_until* 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* macro takes at least 2 thread cycles because it manipulates a 64-bit value representing time. For our computational nodes, 3 timing macros 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 mutexes to ensure a correct ordering of the 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 tedious and brittle. Any change in the code would change the timing of the software, and the NOP insertions would need to be readjusted to ensure the correct number of NOPs are added. Designs now are mostly written in higher level programming languages like C and compiled into assembly code, 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 that each have different execution times, the timing instructions allow us to set the same timing constraints in all nodes regardless of their execution content.

The *delay_and_set* macro only enforces a minimum execution time on the computational nodes. Thus, static timing analysis is still required to verify that the worst-case execution time of all nodes meet the imposed timing constraints set by the application parameters. However, as soon as the timing constraints are met, there are no additional benefits to continue improving the execution speed of the computational nodes; the 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, so no over-provisioning of hardware occurs. We show and present these results 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 [128]. The configuration of the waterhammer example 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 to ensure correctness of functionality of our framework.

The second and third example cover two common diesel injector configurations: the unit pump and the common rail. The data for configuring these examples were taken from reference examples provided by Gamma Technologies' GT-SUITE software package [38]. 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 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

	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

complex where the topology is roughly 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. The GT-SUITE-based models for the comman rail and the unit pump use a 1 cm discretization length, which, using a 1500 m/s wave speed and a stability factor of 0.8, yields a 5.33μs 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 [133] with a speed grade of 3. Each Virtex-6 FPGA logic slice contains 4 LUTs and 8 flip-flops, and this particular 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 [129], and are configured to maximize DSP slice usage and minimize logic slice usage as much as possible. We use the logic slices to synthesize as many cores as possible. For these examples, we use a mapping heuristic that groups nodes requiring the 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 hardware 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 minimum time step we can support. 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 memory access latency is all deterministically one cycle. The PTARM architecture provides deterministic execution times for each instruction implemented, and the full list of instruction execution cycles are listed in table 3.4. 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 are shown in table 4.2.

To convert thread cycles to physical time, we use the processor clock speed and the 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 con-

verts to a physical time of 40 ns. The unit pump and common rail have a time step requirement of 5.33 μs, 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 Δx is used, which leads to a bigger time step than 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 the 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 only; it does not contain any floating point units, and it is not used in our implementation. The baseline configuration used in our implementation is the "basic float", which contains a floating point add/subtracter, a floating point multiplier, and floating point to fixed point 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 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 the architecture of multithreaded processors. Multithreaded 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 the size of the BRAM used for the register sets. The size of the multiplexers used to select thread states and registers are 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 implementations that have sufficient slack time or require faster performance, adjusting the number of threads can help designs meet the required constraints. Our precise execution time analysis allows us to determine the maximum number of threads, six in our case, we need to support to meet our timing constraints. An over estimated execution time in this case could lead to under utilizing the hardware. For example, if we were limited to using five threads for each core in order to meet the timing constraint, then additional cores would be needed to fully implement our 234 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 hardware units will 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 of each configuration.

The actual resource impact can be seen from Table 4.4, which shows the total slices occupied for the three examples after place and route. 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 proves to be critical for the 234-node common rail example, as only our heterogeneous architecture can implement the design on the XC6VLX195T FPGA; the homogeneous design did not fit. These results also reflect our decision to use a heuristic that groups nodes with the similar computations together. By doing so, we synthesize less hardware floating point units, 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. There may exist different implementation options for our application on FPGAs. For example, we could attempt to tackle the problem by using 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 [56] 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 [56]. 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 [25, 31, 137] 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 are all shown to be vulnerabilities that lead to side-channel attacks [20, 87, 7, 30].

In order to not be confused with Kocher's [56] 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 [75, 55], fault injection attacks [22, 36], and many others [137].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [22, 30, 57, 52, 6, 53, 28, 119, 118]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-channel exploits [30, 22, 57, 52, 6, 53, 28, 119, 118] indicate that we do not have the complete set of solutions, as more vulnerabilities are still being discovered and exploited. Just recently, Coppens et al. [30] 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 [56] or blinding signatures [56, 28]. Other techniques such as branch equalization [80, 137] use software techniques to rewrite algorithms such that they take equal time to execute during each conditional branch. We take a different approach to directly address the crux of the problem, which is the *lack of control* over timing behaviors in software. By using the PRET architecture, designed to allow predictable and controllable timing behaviors, we prevent the attacker from exploiting uncontrollable timing side-channel leaks from the architecture.

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 the PRET architecture 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 [57], key-card gates [24], set-top boxes [57], and thumbpods [101], which are a good fit for the PRET architecture's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [82] encryption algorithms on the PRET architecture, 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 outlines a notion of timing attacks [56] 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 can be derived one bit at a time. Kocher also introduced power attacks [75, 55], 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 [81, 53], which also found side-channel attacks against IDEA, RC5 and blowfish [53]. Fault-based attacks [22, 52, 36] were introduced by Bihan et

al. [22]. These attacks attempt to extract keys by observing the system behavior to generated faults. For the side-channel attacks that we have missed, Zhou [137] presents a survey on a wide range of side-channel attacks.

Dhem et al. [31] demonstrate 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 [56, 28], execution time padding [56], and adding random delays [56] have been proposed as possible defenses against this attack. However, these solutions were not widely adopted by the general public until Brumley et al. [25] 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 [36] and AES [20]. In particular, Bernstien's attack on AES [20] 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 [87] 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 [7, 6] that monitor control flow by occupying a shared branched predictor. Compiler and source-to-source transformation techniques [30, 80] have also been developed to thwart side-channel attacks.

Wang et al. [118] 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 [119] and Random Permutation caches [118] that defend against caching attacks in hardware. Very recently, Coppens [30] 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 introduce 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 [56] to shield against the timing variability of the algorithm. At a glance it might seem that the timing instructions 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 the encryption algorithms. The timing instructions, on the other hand, allow for a separation of concerns 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 [87, 6] and obtain side-channel information. We argue that a *predictable* architecture is also required to eliminate timing exploiting attacks.

Predictable Architecture

Pipeline Modern processor architectures often use speculation techniques such as caches and branch predictors to improve average performance. These create *timing variations* in the program execution. Depending on the outcome of its speculation, the processor might need to discard the wrongly speculated work, and re-execute the correct instructions. Since these units are shared by all software processes concurrently running on the processor, the states of the 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 [61]. This makes these units *unpredictable*. 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.

Multithreaded architectures enable more opportunities to exploit the uncontrollable timing behaviors. Attackers exploit such architectures 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 [87, 6] 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 simultaneous multithreading, to be disabled by default in some Linux distributions because of its security risks [88]. For general purpose applications, these side effects pose insignificant threats, but for security applications, the consequences are uncontrollable sources of side-channel information leakages.

As discussed in chapter 2.1.3, PRET employs a thread-interleaved pipeline, a multithreaded pipeline that uses a predictable round-robin thread scheduling policy between the hardware threads every cycle. The thread-interleaved pipeline eliminates the need for any data forwarding/bypassing logic, along with the need for hardware speculation units such as branch predictors. 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. Most importantly, the hardware threads are temporally isolated, meaning that no threads can affect each others timing behavior. This prevents attackers from exploiting shared resources from the pipeline to initiate timing side-channel attacks.

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 [112]. Caches abstract away the memory hierarchy and the access latency variation from the programmer by managing the cache contents in 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 the eviction causes [87]. 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. 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. This prevents unwanted timing side-effects from attackers and spy threads, even if 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 among 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 provides control of timing properties in software and a predictable architecture that with temporal isolation for its hardware threads. PRET is impenetrable known attacks such as branch predictor attacks [7], cache attacks [87] or other attacks on the pipeline [30]. More impor-

tantly, 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 show the results of two encryption algorithms running on the cycle accurate simulator of the PRET architecture described in lickly et al [67]. Lickly et al.[67] introduced the first realization of the PRET architecture that implements the SPARC v8 instruction set. It employs six threads on a six stage thread-interleaved pipeline, and also uses scratchpads for an expose memory hierarchy. Programs are written in C and compiled using a standard gcc cross compiler from Gaisler research labs [37]. Because the results of these experiments have yet to be ported to the newer PTARM, we present the original measurements obtained on the SPARC realization of the PRET architecture.

The precision timed SPARC architecture implements a simple processor extension inspired by Ip and Edwards [49] that adds timing instructions to the ISA. To be consistent with the terminology used in [49], we call this instruction the *deadline instruction*. This deadline instruction has similar semantics to the *delay_and_set* macro 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 the units of thread cycles, which are 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 [56].

```
   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
```

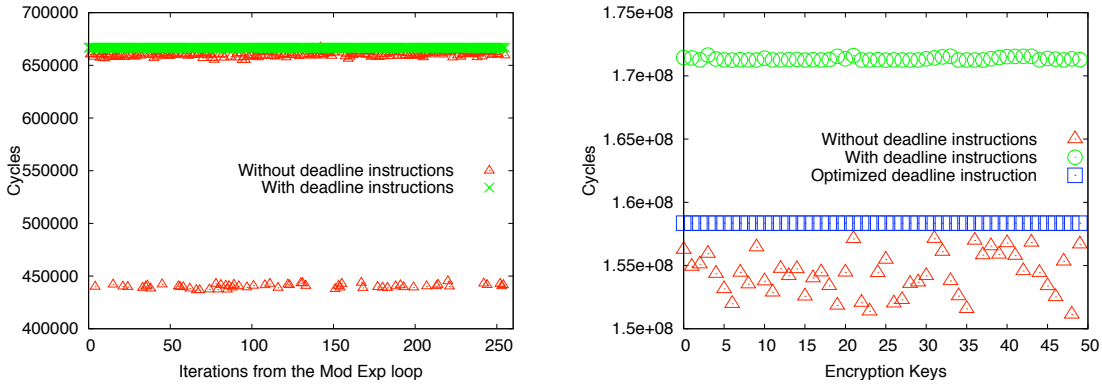
Algorithm 1: RSA Cipher

```
   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 precision timed SPARC 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.



(a) Run time of Modular Exponent operation

(b) Run time of RSA operation

Figure 4.8: RSA Algorithm

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.

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 have showed. Although this makes the encryption algorithm completely secure against time-exploiting attacks, this method is not adopted in practice because of the 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 [56]. 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 ± 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.

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 cryptographically strong [94]. 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 ±2 standard deviation are the least secure keys, 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.

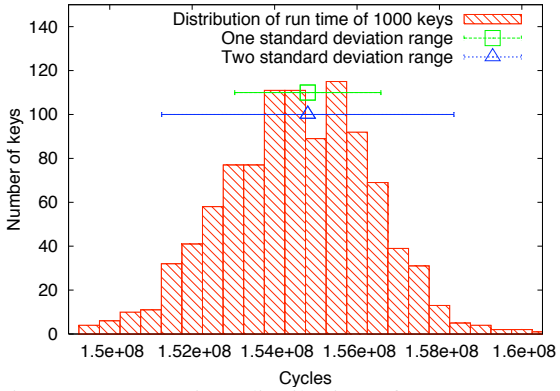
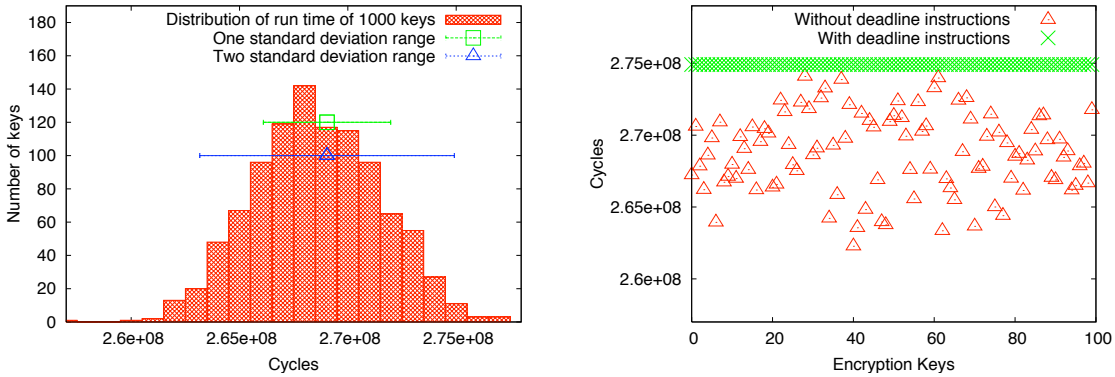


Figure 4.9: Run time distribution of 1000 randomly generated keys for RSA

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 obtain the overhead introduced for each of the keys with run time below 2 standard deviations (97.9% of keys in our case). 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.

Digital Signature Algorithm

Kocher's [56] original paper mentioned that the Digital Signature Standard [82] is also susceptible to timing attacks. Thus, to further illustrate our case, we port the Digital Signature Algorithm from the current OpenSSL library (0.9.8j) onto the precision timed SPARC architecture. We use 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 generate another one hundred keys, and measure the run time with and without deadline instructions, which we show in figure 4.10(b). We can clearly observe that the run time with deadline instructions is constant, and any time-exploiting attack is not possible.



(a) Distribution of 1000 DSA keys

(b) Run time of 100 DSA operations

Figure 4.10: Digital Signature Standard Algorithm

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 work 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 removes timing interference between threads through a thread-interleaved pipeline, scratchpad memories, and a predictable DRAM memory controller. This eliminates the shared states in the architecture that create uncontrollable timing interference,

exploited by the 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 that are less researched upon.

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 an 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 mainly 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 at the ISA level.

5.1 Pipeline-Focused Techniques

5.1.1 Static Branch Predictors

Dynamic branch predictors cause timing anomalies [34], and they are difficult to model 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. [26] make a case for *static branch prediction* to be used for real-time systems. This can 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 scheme uses the observation that for loop control branches, almost all backwards branches are taken to return to the loop; 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. [23] use this idea of software static branch prediction to improve the 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 can modify the cache state. However, the experiments assumed all code and data fit into the caches, thus the effects of caches were not factored into 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. [98] 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 can be significantly reduced, as each basic block will start with a consistent initial state. However, the results assume the absence of caches, which can 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 can induce large penalties for programs with a lot of control flow transfer and small basic blocks.

Whitham et al. [123] combine the techniques of instruction pre-scheduling and static branch predictions to 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*[122]. *Virtual traces* are program paths with static branch predictions inserted. These are usually formed by predicting along the WCEP, similar to the algorithm introduced by Bodin et al [23]. 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. 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 and reordering of branches. The architecture employs scratchpads instead of caches. This allows the execution of traces to run predictably for each different exit (branch mispredict) within a trace. The architecture shows an improved throughput for most programs when compared to a simple in-order CPU model.

5.1.3 VLIW architectures

VLIW machines, like superscalars, issue multiple instructions at a time to exploit ILP. However, unlike superscalars, VLIW machines rely on the compiler to utilize ILP and determine the instructions issued. This helps in the predictability because the hardware does almost no reordering or stalling.

Yan et al. [135] study the predictability of VLIW machines, and propose changes to the architecture and compiler to improve the predictability. They find 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, since statically it is not known whether a memory access is a hit

or a miss, the hardware still needs to check for it and stall when needed. Second, data dependencies still exist across compilation units, so the hardware still needs to support basic data dependency checking to handle those dependencies. A compilation unit could be a basic block, a loop, a procedure or a region [41]. Finally, if the VLIW machine uses branch prediction, there is still the need for the handling of mispredictions.

As VLIW machines heavily utilize the compiler to improve performance, they propose several compiler techniques to compile programs that lend themselves to better WCET. First, they use the single-path paradigm proposed by Puschner and Burns [91], and eliminate all non-loop backwards branches with full if-conversions [12]. To mitigate the performance penalty of single-path programming, aggressive hyperblock scheduling [71] is used 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 for 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. [59] evaluate the use of different real time scheduling schemes to schedule hardware threads to handle external events. They evaluate fixed priority preemptive (FPP), earliest deadline first (EDF), least laxity first (LLF) and guaranteed percentage (GP), which is similar to time sharing the pipeline. The architecture used for evaluation is a Java multithreaded superscalar pipeline with four threads [58]. A hardware priority manager is implemented to facilitate the scheduling of threads. All real-time threads register their real-time requirements during initialization with 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 to 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. [33] propose 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 processor. 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. In 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 involve slight modifications to architecture to create a *WCET-aware* SMT to be used for real-time systems.

Barre et al. [18] propose 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. [47] show that out-of-order execution may not be as cost effective as in-order execution on SMT machines. Thus, Uhrig et al. [100] propose a similar concept to Barre et al. [18], except for an in-order executed superscalar. Mische et al.[78] expand this to allow more than one real-time thread to run on the SMT architecture. This is done by time-sharing the highest priority thread slot among 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. However, it is unclear how contention between the real-time threads on the high priority thread slot is resolved.

Thread Interleaved Pipelines

Thread-interleaved pipelines have been proposed and employed in various architectures from research and industry. Besides the CDC6600 [2], described in section 2.1.3, Lee and Messerschmitt [62], the Denelcore HEP [107], the XMOS XS1 architecture [72], the Parallex Propeller chip [4] and the Sandbridge Sandblaster [35] all use fine grained thread interleaving for different applications. In particular, Lee and Messerschmitt [62] and the Sandbridge Sandblaster [35] propose the use of thread-interleaved pipelines for DSP applications. Lee and Messerschmitt [62] also

use a round robin thread scheduling policy while the Sandblaster uses a Token Triggered Threading policy. The Token Triggered Threading policy is similar to the round robin scheduling policy in that each hardware thread context can only issue one instruction each in a scheduling cycle. However, a token is used to determine which thread's instruction is executed next. The XMOS XS1 architecture [72] allows hardware threads to dynamically be added and removed from the thread scheduling. They use the dynamically added threads to handle interrupts, which improves the interrupt response latency. The XS1 architecture specifies that during execution, there will always be a minimum the number of threads equal to the pipeline depth. As explained in section 2.1.3, this removes pipeline hazards to improve throughput. However, the dynamic thread scheduling can cause each thread's execution frequency to vary depending on the number of threads executing at one time.

5.1.5 Others

Virtual Simple Architecture

Anantaraman et al. [13] propose the virtual simple architecture (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 timing of a hypothetical simple pipeline that is amenable to safe and tight WCET analysis. And a high performance mode, in 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 executing 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 in attempt to meet the timing constraints. The results show that the high performance mode have average executions times of 3 to 4 times faster than the simple mode. The authors also discuss 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 whether the simple mode will always be able to make up the time if the high performance mode misses its checkpoint.

Java Optimized Processor

Schoeberl presents the Java Optimized Processor (JOP) [104] which uses Java for real time embedded systems. The design of JOP includes a two level stack cache architecture [103]. Instead of using a large register file to store the stack, like the PicoJava[74], it uses only two registers to store the top two entries of the stack (Register A and Register 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 number 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 introduces the Microprogrammed Coarse Grained Reconfigurable Processor (MCGREP) [121], 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 other's execution time. A fast internal RAM without cache is used to store the program and 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.

ARPRET

Andalam et al. [14] introduce the Auckland Reactive PRET (ARPRET) architecture to execute a new language called precision timed C (PRET-C). PRET-C is a synchronous language extension to C designed to support synchronous concurrency, preemption, and a high-level construct for logical time. ARPRET extends the Microblaze [130] with a custom Predictable Functional Unit (PFU) that is used for thread scheduling. The Microblaze is configured to use on-chip memory to achieve predictable memory access latencies. The PFU stores the thread contexts of each thread, including the PC, thread status, priority, etc. that are used during each context switch. By doing the thread scheduling in hardware, ARPRET reduces the thread switching overhead. Each thread switch is triggered in software by the C language extensions in PRET-C, and the PFU is used to determine the next context to run. Their benchmarks show that the ARPRET achieves predictable execution without sacrificing throughput.

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. [95] 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. [43]. The results confirm that the LRU replacement policy was significantly more predictable than other policies. Thus, the authors recommend any real-time system with caches to use LRU for its replacement policy. This paper also reveals potential for improvement in existing analyses of PLRU and FIFO.

Puat and Decotigny [89] propose 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 when

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 [102] propose to use method caches for the instruction cache of the JOP architecture [104]. 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 tradeoff 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. [76] use a method cache mechanism with the real-time SMT architecture in [78]. They partition the scratchpad for each different thread so no inter-thread interference will exist. Then, they implement the method cache [54] 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 [120] because the contents are managed in software. Puaut et al. [90] present a comparison of locked caches and scratchpads, and show only subtle differences between the two in terms of performance. Most benchmarks give 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 [124] introduce 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 stall other DRAM accesses, and potentially close 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 [21] tackle this specific issue of DRAM refreshes by scheduling burst refreshes. They account for the DRAM refresh requirements in software, and schedule refresh tasks to handle the 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. [9, 10, 8] introduce the Predator, a predictable SDRAM memory controller. It is predictable by providing a guaranteed maximum latency and minimum bandwidth to each client, independent of the behavior of other clients. 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. [85] present 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

In order to improve the efficiency and scalability of handing *time* in CPS and safety critical systems, we contend that changes must be made to conventional abstraction layers to introduce *time* as a first class citizen. In this thesis we focus on doing so for the ISA abstraction layer and below. We explore instruction extensions to the ARM ISA to bring temporal semantics to the program, independent of architecture implementation. We also present the precision timed ARM (PTARM), an implementation of a PRET machine, in order to provide a timing predictable and composable platform for deterministic execution times.

To bring temporal semantics to the ISA abstraction layer, we present a few instruction extensions to the existing instruction set. The instructions operate on a platform clock that is synchronous with the execution of instructions. The instruction set extensions allow programmers to specify the timing properties of program segments, and to throw hardware exceptions when the timing specifications are not met. In this way, our instruction extensions do not over constrain the temporal semantics of the ISA, and continue to allow architecture innovation to improve program performance. These extensions allow programmers to begin reasoning about temporal properties of the programs independent of the underlying execution platform, provided that the ISA is faithfully implemented.

The PTARM exploits thread-level parallelism for performance by employing a predictable thread-interleaved pipeline. This removes the unpredictability when handling pipeline hazards, and provides temporal isolation for all hardware threads within the pipeline. PTARM uses scratchpads instead of caches to expose the memory hierarchy, which enables a simpler and more precise WCET analysis of memory accesses. With a bank privatized DRAM controller, PTARM provides predictable DRAM access latencies for each hardware thread, and preserves temporal isolation between the hardware threads that access the DRAM as a shared resource. The timing predictability and composability provided by PTARM does not come at the cost of aggregate performance, as our benchmarks show an improved throughput for both the pipeline and DRAM memory controller when fully utilized. Although achieving full utilization requires that applications have sufficient concurrency, the deterministic architecture can better equip CPS platforms with the ability to handle the concurrency and the uncontrollable timing properties exhibited by physical processes.

We also demonstrate the benefits of a PRET machine in the context of a real-time engine fuel rail simulator and embedded security. To simulate an engine fuel rail in real time, we implement a platform that uses multiple PTARM cores that communicate through local shared buffers. The

predictable timing of PTARM allows us to statically verify that the timing constraints are met. The timing control provided by the extended ISA enables us to implement a software based low overhead time synchronized communication protocol between the hardware threads, saving the resources required to implement a full hardware interconnection system. These features of PTARM enable us to implement a scalable solution that can simulate, in real time, a 237 node common fuel rail systems in a single Xinlinx Virtex 6 FPGA. In the context of embedded security, the underlying architectures implementing encryption algorithms are susceptible to timing side-channel attacks. Attackers can exploit the uncontrollable execution time variances caused by the architecture or algorithm to derive the secret key. We implement the RSA and DSA encryption algorithms on a PRET architecture, and show that a predictable architecture with controllable timing properties in the ISA not only defends against all timing related side-channel attacks, but eliminates the root cause of them.

We continue to investigate the several challenges and questions mentioned in this thesis. First, we continue to explore the formalization of the timing extensions to the ISA. The introduction of temporal semantics in the ISA should be platform independent; our implementation in PTARM merely opens up opportunities for further experimentation and research. Nailing down the formal semantics of each instruction extension is key to a consistent meaning of "time" that is independent of the underlying hardware implementation. Second, we continue to experiment with how a predictable pipeline and memory controller handles external interrupts and I/O devices. With the plethora of complex interfaces and protocols for modern high speed I/O interactions, typical I/O controllers are implemented in hardware. However, we envision that a predictable architecture with precise timing control can enable software implementations of protocols typically implemented in hardware, due to the lack of precise control over timing in software. A software implementation can enable flexibility for different protocols and reduce design efforts, leading to faster time-to-market and more feature rich designs. Third, we continue to explore the interfacing with a timing predictable bus or interconnect, which can be used in timing predictable multicore systems. In our real-time engine fuel rail simulator, we show a multicore implementation with PRET architectures that uses local shared memories for communication and a timing based synchronization communication protocol implemented in software. However, as communication schemes and applications become more complex, the interconnect or bus will play a more integral role in the connection and communication of multiple PRET cores. Thus, our future work also includes predictable communication protocols across interconnects and shared buses that leverage the predictable timing of the PRET architecture.

It is important to understand that we are not proclaiming that all dynamic behavior in systems are harmful. However, the dynamic behavior must be controllable. For example, dynamically scheduling hardware threads in the architecture causes uncontrollable timing interference because the triggering of thread switches is hidden from, and cannot be explicitly controlled by, the programmer. We argue that only by achieving predictability in the architecture and platforms can we begin to reason about more dynamic behavior in software. With a predictable architecture and the introduction of temporal semantics in the ISA, we hope to provide a timing deterministic foundation in the lower levels of abstraction. In doing so, we enable larger and more complex designs of cyber-physical systems to gain more precise and efficient control over the timing properties of the system.

Bibliography

[1] Autosar (automotive open system architecture). http://www.autosar.org/.

[2] Control data 6400/6500/6600 computer systems reference manual. http://ed-thelen.org/comp-hist/CDC-6600-R-M.html.

[3] GNU ARM Toolchains. http://www.gnuarm.com/.

[4] Parallex propeller chip website.

[5] Ch7301c dvi transmitter device, May 2005. http://www.xilinx.com/products/boards/ml505/datasheets/7301ds.pdf.

[6] 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.

[7] 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.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

[14] S. Andalam, P. S. Roop, and A. Girault. Predictable multithreading of embedded applications using pret-c. In *MEMOCODE*, pages 159–168, 2010.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

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

[29] 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.

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

[31] 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.

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

[33] 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.

[34] 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.

[35] J. G. Erdem. Multi-threaded processor for software-defined radio, 2002.

[36] 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.

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

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

[39] 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.

[40] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. pages 137–147, Brussels, Belgium, July 2010. OCG.

[41] 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.

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

[47] 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.

[48] Intel. *Intel StrongARM SA-1100 Microprocessor - Developer's Manual*, April 1999.

[49] 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.

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

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

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

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

[58] 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.

[59] 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.

[60] 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.

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

[62] 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.

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

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

[65] 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.

[66] K. Lee and J. Eidson. Ieee-1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *In 34 th Annual Precise Time and Time Interval (PTTI) Meeting*, pages 98–105, 2002.

[67] 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.

[68] I. Liu, E. A. Lee, M. Viele, G. G. Wang, and H. Andrade. A heterogeneous architecture for evaluating real-time one-dimensional computational fluid dynamics on fpgas. In *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), Toronto, Canada*, April 2012. Following section 8.1.9 (pg. 56) of the IEEE PSPB Operations Manual we make the following statement: "This work has been submitted to the IEEE for possible publication. Copyright may be transferred without notice, after which this version may no longer be accessible.".

[69] 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.

[70] 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.

[71] 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.

[72] D. May. *The XMOS XS1 Architecture*. XMOS, October 2009.

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

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

[75] 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.

[76] 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.

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

[78] 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.

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

[80] 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.

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

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

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

[84] 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.

[85] 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.

[86] 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.

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

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

[89] 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.

[90] 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.

[91] 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.

[92] W. Qin and S. Malik. Flexible and formal modeling of microprocessors with application to retargetable simulation. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '03, pages 10556–, Washington, DC, USA, 2003. IEEE Computer Society.

[93] J. W. Ramsey. Integrated modular avionics: Less is more. Avionics Magazine, February 2007. http://www.aviationtoday.com/av/categories/commercial/8420.html.

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

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

[96] 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.

[97] 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.

[98] 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.

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

[100] 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.

[101] 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.

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

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

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

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

[106] 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.

[107] B. Smith. The architecture of hep. In *on Parallel MIMD computation: HEP supercomputer and its applications*, pages 41–55, Cambridge, MA, USA, 1985. Massachusetts Institute of Technology.

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

[109] 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.

[110] 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.

[111] 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.

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

[113] 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.

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

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

[116] 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.

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

[118] 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.

[119] 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.

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

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

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

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

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

[125] 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.

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

[127] 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.

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

[129] Xilinx. *Core generator guide*.

[130] Xilinx. Microblaze soft processor core. http://www.xilinx.com/tools/microblaze.htm.

[131] Xilinx. Platform studio and the embedded development kit (edk). http://www.xilinx.com/tools/platform.htm.

[132] Xilinx. Virtex-5 family overview, February 2009. http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf.

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

[134] Xlinx. Xilinx ml505 evaluation platform documentation. http://www.xilinx.com/products/boards/ml505/docs.htm.

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

[136] 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.

[137] Yongbin. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. http://eprint.iacr.org/2005/388.

[138] Y. Zhao, J. Liu, and E. A. Lee. A programming model for time-synchronized distributed real-time systems. In *13th IEEE Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07*, pages 259 – 268, April 2007.