Precision Timed Machines

by

Isaac Liu

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

in

Electrical Engineering and Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY OF CALIFORNIA, BERKELEY

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

Spring 2012

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

Chair Date

Date

Date

University of California, Berkeley

Spring 2012

Precision Timed Machines

Abstract

Precision Timed Machines

by

Isaac Liu
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Edward A. Lee, Chair

This is my abstract

To my wife Emily Cheung, my parents Char-Shine Liu and Shu-Jen Liu, and everyone else whom I've had the privilege of running into for the first twenty-seven years of my life.

Contents

List of Figures

List of Tables

Acknowledgments

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

Chapter 1

Introduction

Outline
(Todo: make sure to add in timing anomalies)

1.1 Background

- Discuss the problem
- show the difficulty in execution time analysis of a simple c code
- show the variability different architecture improvements have introduced to improve average case execution time (Sami's graph)

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

1.2 Intro Section Header 2

Here is another header
Talk about timing anoma

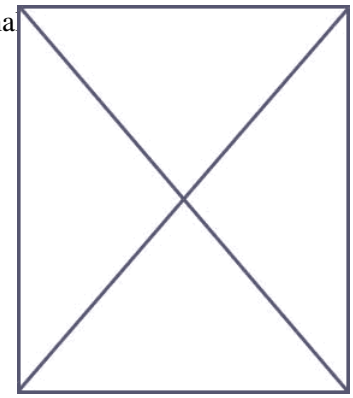


Figure 1.1: Image Placeholder

The remaining chapters are organized as follows. Chapter 5 surveys the related research that has been done on architectures to make them more analyzable. Chapter 2 explains the architecture of PRET including the thread-interleavedpipeline and memory hierarchy, Chapter **??**, Chapter 4, Chapter 6,

Chapter 2

Precision Timed Machine

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

2.1 Pipelines

The introduction of pipelining vastly improved the average-case performance of programs. It allows faster clock speeds, and improves instruction throughput compared to single cycle architectures. Pipelining begin executing subsequent instructions while prior instructions are still in execution. Ideally each processor cycle one instruction completes and leaves the pipeline as another enters and begins execution. In reality, different pipeline hazards occur which reduce the throughput and create stalls in the pipeline. Different techniques were introduced to handle the effects of pipeline hazards, and greatly effect to the timing predictability and repeatability of an architecture. To illustrate this point, we discuss some basic hardware additions proposed to reduce performance penalty from hazards, and show how they effect the execution time and predictability.

2.1.1 Pipeline Hazards

Data hazards occur when instructions need the results of previous instructions that have not yet committed. The code segment shown in figure 2.1 contains instructions that each depend on the result of its previous instruction. Figure 2.2 shows two ways data hazards can be handled in a single-threaded pipeline. In the figure, time progresses horizontally towards the right, each time step, or 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

```
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

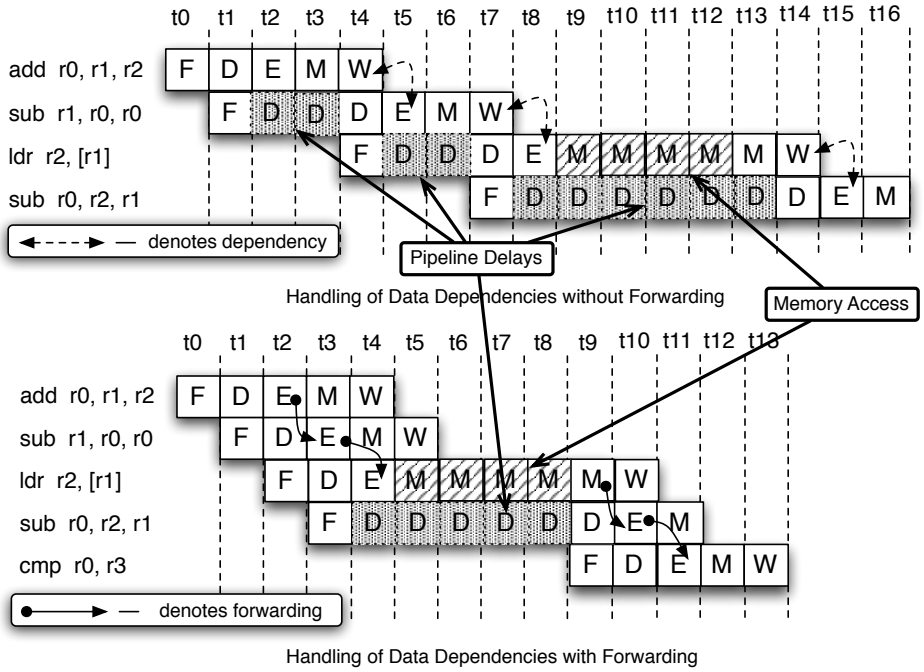


Figure 2.2: Handling of data dependencies in single threaded pipelines

the pipeline – fetch (F), decode (D), execute (E), memory (M) and writeback (W). The pipelines here are assumed to have a similar design to the five stage pipeline mentioned in Hennessy and Pattern (Todo: cite hennessy and patterson).

A simple but effective way of handling data hazards is by simply stalling the pipeline until the previous instruction completes. This is shown in the top of figure 2.2. Pipeline delays (or bubbles) are inserted for instructions to wait until the previous instruction is complete. The dependencies between instructions are shown in the figure to make clear why the pipeline bubbles are necessary. The performance penalty incurred in this case is the pipeline delays inserted to wait for the previous instruction to complete. Data forwarding was introduced to remove the need for inserting bubbles into the pipeline. Data forwarding relies on the fact that the results of the previous instruction is typically available before the it commits. A data forwarding circuitry consists of backwards paths for data from later pipeline stages to the inputs of earlier pipeline stages, and multiplexers to select amongst all data signals. Because it provides a way to directly access computation results from the previous instruction before the previous instruction finishes, it removes the need to wait for the previous instruction to commit. The pipeline controller dynamically detects whether a data-dependency exists, and changes the selection bits to the multiplexers accordingly so the correct operands are selected. The bottom of figure 2.2 shows the execution with forwarding in the pipeline. No pipeline bubbles are needed for the first *sub* instruction and *ld* instruction because the data they depend on are forwarded with the forwarding paths. However, the second *sub* instruction after the *ld* instruction still stalls. As mentioned earlier, forwarding relies on the results from the previous instruction being available before the previous instruction commits. In the case of longer latency operations, such as memory accesses, the data cannot be forwarded until it becomes available, so stalls are still required. The memory access latency in the figure is arbitrarily chosen to be 5 cycles

so the figure is not too long and instructions after the *ld* instruction can be shown. We purposely leave out the details regarding memory accesses at this point, and will discuss it extensively in section 2.2. We merely use the *ld* instruction to illustrate the limitations of data forwarding. They can address the data-dependencies caused by pipelining – the read-after-write of register computations. However, they cannot address the data-dependencies caused by other long latency operations such as memory operations, so pipeline stalls are still needed. More involved techniques such as the introduction of out-of-order execution or superscalar pipelines are used to mitigate the effects of long latency operations. We will discuss more of these in chapter 5 when we mention the related works.

To understand the timing effects of handling data hazards, we discuss how to determine execution time for instructions using both methods of handling data hazards. With the simple method of inserting stalls, we need to know when the stalls will be inserted and how long the instruction will need to stall for. This information can be determined by simply checking the previous instruction since stalls are inserted only if this instruction depends on the results of the previous instruction. Within pipelines, the execution of most instructions are deterministic, so for the most part we can determine how long the stall will be by checking the previous instruction. Memory access instructions are an exception to instructions that have deterministic execution time, but as mentioned before, we will discuss these extensively in section 2.2. For pipelines with data forwarding, we need to know in what situations the data forwarding circuitry cannot correctly forward the data to the next instruction. Although the pipeline dynamically forwards the data during run-time, the logic in the pipeline controller that enables and selects the correct forwarding bits only needs to keep track of a small set of previous instructions to detect data-dependencies. The set of instructions it needs to check usually depends on the depth of the pipeline. Thus, static execution time analysis can detect forwarding by simply checking a short window of previous instructions to account for stalls accordingly. (Todo: find papers to back this up) We simplified greatly the execution time analysis discussed above to ignore effects from other pipeline mechanisms. We wanted to simply focused on the effects of handling data-hazards through stalling or data-forwarding. We can see that both methods of handling data-hazards cause instruction execution time to depend previous instruction execution history. But the execution history that instruction execution time is dependent upon is small and temporary enough to be accounted for.

Branches cause control-flow hazards in the pipeline; the instruction after the branch, which should be fetched the next cycle, is unknown until after the branch instruction is completed. Conditional branches further complicates matters, as whether or not the branch is taken depends on an additional condition that could possible be unknown when the conditional branch is in execution. The code segment in figure 2.3 shows assembly instructions from the ARM instruction set architecture (ISA) that implement the Greatest Common Divisor (GCD) algorithm using conditional branch instructions *beq* (branch equal) and *blt* (branch less than). Conditional branch

```
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: Sample code for GCD with conditional branches

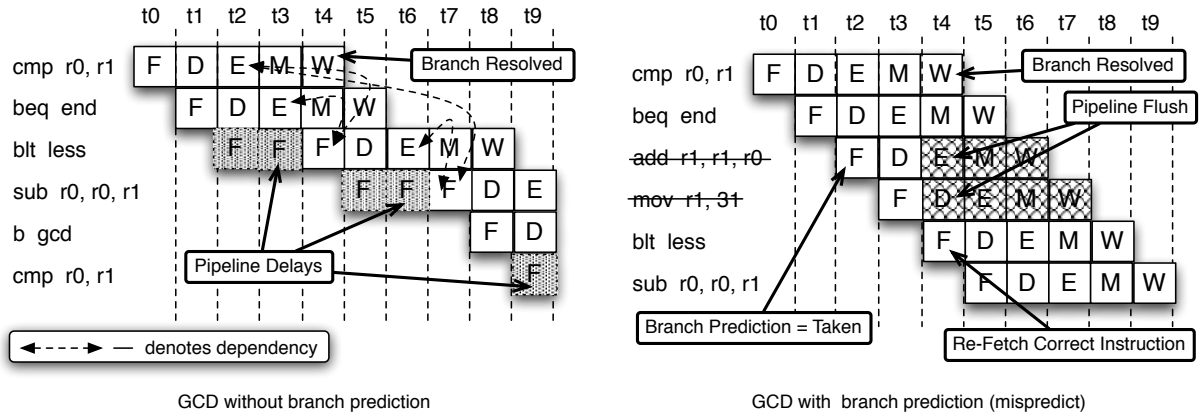


Figure 2.4: Handling of conditional branches in single threaded pipelines

instructions in ARM branch based on conditional bits that are stored in a processor state register and set with special compare instructions(Todo: cite arm manual). The *cmp* instruction is one such compare 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 show two ways branches can be handled in a single-threaded pipeline.

Similar to handling data-hazards, a simple but effective way of handling control-flow hazards is by simply stalling the pipeline until the branch instruction completes. This is shown on the left of figure 2.4. Two pipeline delays (or bubbles) are inserted after each branch instruction to wait until address calculation is completed. The dependencies between instructions are also drawn out to make clear why the pipeline bubbles are necessary. In order for the *blt* instruction to be fetched, its address must be calculated during the execution stage of the *beq* instruction. At the same time, because *beq* is a conditional branch, whether or not the branch is taken depends on the *cmp* instruction. The pipeline here is assumed to have forwarding circuitry, so the addresses calculated by the branch instructions and the results of the *cmp* instruction can be used before the instructions are committed. The performance penalty incurred is the pipeline delays inserted to wait for the branch address calculation to complete. Conditional branches will also incur extra delays for deeper pipelines if the branch condition cannot be resolved in time. Some architectures enforce the compiler to insert one or more non-dependent instructions after a branch that is always executed before the change in control-flow of the program. These are called branch delay slots and can mitigate the branch penalty, but become less effective as pipelines grow deeper because the longer delay slots are required.

In attempt to remove the need of inserting pipeline bubbles, branch predictors were invented to predict the results of a branch before it is resolved(Todo: citation). Many clever branch predictors have been proposed, and they can accurately predict branches up to 93.5%(Todo: citation). Branch predictors predict the condition and target addresses of branches, so pipelines can speculatively continue execution based upon the prediction. If the prediction was correct, no penalty occurs for the branch, and execution simply continues. However, when a mispredict occurs, then

the speculatively executed instructions need to be flushed and the correct instructions need to be refetched into the pipeline for execution. The right of figure 2.4 shows the execution of GCD in the case of a branch misprediction. After the *beq* instruction, the branch is predicted to be taken, and the *add* and *mov* instructions from the label *end* is directly fetched into execution. When the *cmp* instruction is completed, a misprediction is detected, so the *add* and *mov* instruction are flushed out of the pipeline while the correct instruction *blt* is immediately re-fetched and execution continues. The misprediction penalty is typically the number of stages between fetch and execute, as those cycles are wasted executing instructions from an incorrect execution path. This penalty only occurs on a mispredict, thus branch prediction typically yields better average performance and is preferred for modern architectures. Nonetheless, it is important to understand the effects of branch prediction on execution time.

Typical branch predictors predict branches based upon the history of previous branches encountered. As each branch instruction is resolved, the internal state of the predictor, which stores the branch histories, is updated and used to predict the next branch. This implicitly creates a dependency between branch instructions and their execution history, as the prediction is affected by its history. In other words, the execution time of a branch instruction will depend on the branch results of previous branch instructions. During static execution timing analysis, the state of the branch predictor is unknown because is it often infeasible to keep track of execution history so far back. There has been work on explicitly modeling branch predictors for execution time analysis(Todo: citation), but the results are (Todo: the results of branch predictor modeling for execution time analysis). The analysis needs to conservatively account for the potential branch mispredict penalty for each branch, which leads to overestimated execution times. To make matters worse, as architectures grow in complexity, more internal states exist in architectures that could be affected by the speculative execution. For example, cache lines could be evicted when speculatively executing instructions from a mispredicted path, changing the state of the cache. This makes a tight static execution time analysis extremely difficult, if not impossible; explicitly modeling all hardware states and their effects together often lead to an infeasible explosion in state space. On the other hand, although the simple method of inserting pipeline bubbles for branches could lead to more branch penalties, the static timing analysis is precise and straight forward, as no prediction and speculative execution occur. The timing analysis simply adds the branch penalty to the instruction after a branch. Additional penalties from a conditional branch can be accounted for by simply checking for instructions that modify the conditional flag above the conditional branch. We explicitly showed this simple method of handling branches to point out an important trade-off between speculative execution for better average performance and consistent stalling for better predictability. Average-case performance can be improved by speculation at the cost of predictability and potentially prolonging the worst-case performance. The challenge remains to maintain predictability while improving worst-case performance, and how pipeline hazards are handled play an integral part of tackling this challenge.

2.1.2 Pipeline Multithreading

Multithreaded architectures were introduced to improve instruction throughput over instruction latency. The architecture optimizes thread-level parallelism over instruction-level parallelism to improve performance. Multiple hardware threads are introduced into the pipeline to fully utilize thread-level parallelism. When one hardware thread is stalled, another hardware thread can be fetched into the pipeline for execution to avoid stalling the whole pipeline. To lower the context

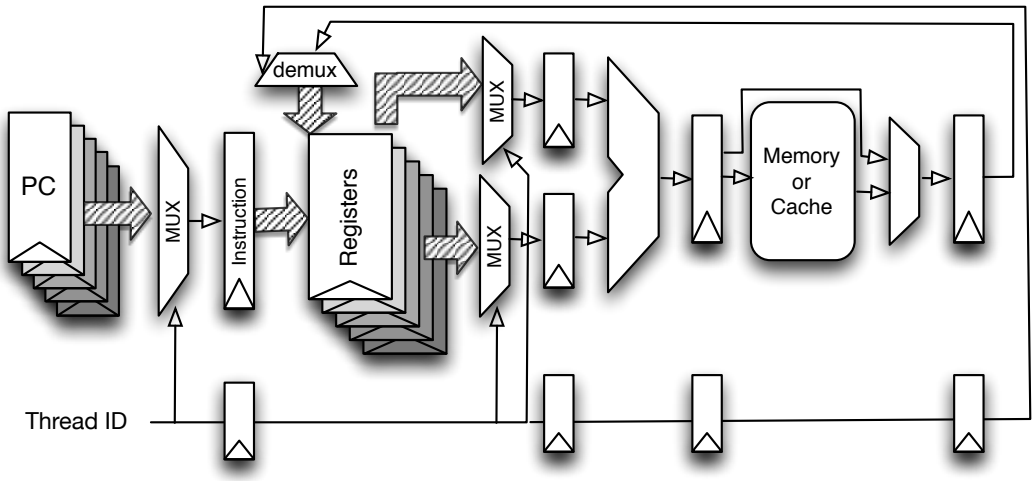


Figure 2.5: Simple Multithreaded Pipeline

switching overhead, the pipeline contains physically separate copies of hardware thread states, such as registers files and program counters etc, for each hardware thread. Figure 2.5 shows a architectural level view of a simple multithreaded pipeline. It contains 5 hardware threads, so it has 5 copies of the Program Counter (PC) and Register files. Once a hardware thread is executing in the pipeline, its corresponding thread state can be selected by signaling the correct selection bits to the multiplexers. The rest of the pipeline remains similar to a traditional 5 stage pipeline as introduced in Hennessy and Pattern(Todo: citation). The extra copies of the thread state and the multiplexers used to select them thus contribute to most of the hardware additions needed to implement hardware multithreading.

Ungerer et al. [90] surveyed different multithreaded architectures and categorized them based upon the (Todo: thread selection?) policy and the execution width of the pipeline. The thread selection policy is the context switching scheme used to determine which threads are executing, and how often a context switch occurs. Coarse-grain policies manage hardware threads similar to the way operation systems manage software threads. A hardware thread gain access to the pipeline and continues to execute until a context switch is triggered. Context switches occur less frequently via this policy, so less hardware threads are required to fully utilize the processor. Different coarse-grain policies trigger context switches with different events. Some trigger on dynamic events, such as cache miss or interrupts, and some trigger on static events, such as specialized instructions. Fine-grain policies switch context much more frequently – usually every processor cycle. Both coarse-grain and fine-grain policies can also have different hardware thread scheduling algorithms that are implemented in a hardware thread scheduling controller to determine which hardware thread is switched into execution. The width of the pipeline refers to the number of instructions that can be fetched into execution in one cycle. For example, superscalar architectures have redundant functional units, such as multipliers and ALUs, and can dispatch multiple instructions into execution in a single cycle. Multithreaded architectures with pipeline widths of more than one, such as Sumultanous Multithreaded (SMT) architectures, can fetch and execute instructions from several hardware threads in the same cycle.

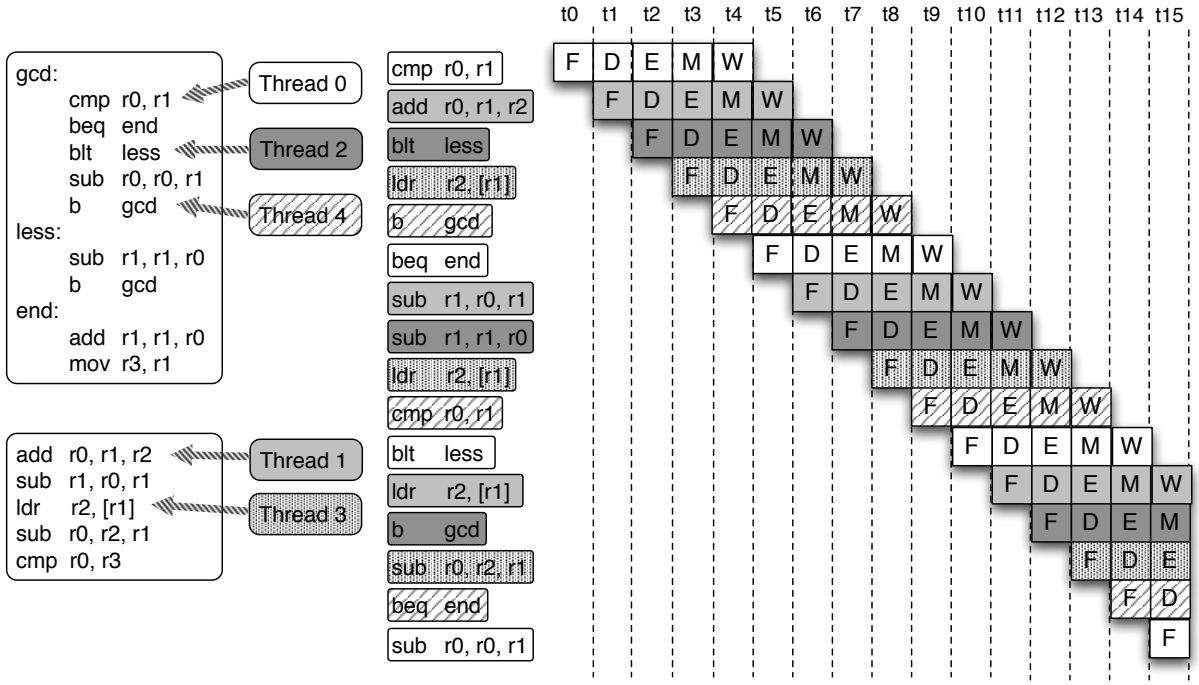


Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

Multithreaded architectures typically bring additional challenges to execution time analysis of software running on them. Any timing analysis for code running on a particular hardware thread needs to take into account not only the code itself, but also the thread selection policy of the architecture and sometimes even the execution context of code running on other hardware threads. For example, if dynamic coarse-grain multithreading is used, then a context switch could occur at any point when a hardware thread is executing in the pipeline. This not only has an effect on the control flow of execution, but also the state of any hardware that is shared, such as caches or branch predictors. Thus, it becomes nearly impossible to estimate execution time without knowing the exact execution state of other hardware threads and the state of the thread scheduling controller. However, it is possible to for multithreaded architectures to fully utilize thread-level parallelism while still maintaining timing predictability. Thread-interleaved pipelines use a fine-grain thread switching policy with round robin thread scheduling to achieve high instruction throughput while still allowing precise timing analysis for code running on its hardware threads. Below, its architecture and trade-offs are described and discussed in detail along with examples and explanation of how timing predictability is maintained. Through the remainder of this chapter, we will use the term "thread" to refer to explicit hardware threads that have physically separate register files, program counters, and other thread states. This is not to be confused the common notion of "threads", which is assumed to be software threads that is managed by operating systems with thread states stored in memory.

2.1.3 Thread-Interleaved Pipelines

The thread-interleaved pipeline was introduced to improve the response time of handling multiple I/O devices (Todo: citation). I/O operations often stall from the communication with the I/O devices. Thus, interacting with multiple I/O devices leads to wasted processor cycles that are idle waiting for the I/O device to respond. By employing multiple hardware thread contexts, a hardware thread stalled from the I/O operations does not stall the whole pipeline, as other hardware threads can be fetched and executed. Thread-interleaved pipelines use fine-grain multithreading; every cycle a context switch occurs and a different hardware thread is fetched into execution. The threads are scheduled in a deterministic round robin fashion. This also reduces the context switch overhead down to nearly zero, as no time is needed to determine which thread to fetch next. Barely any hardware is required to implement round robin thread scheduling; a simple $log(n)$ bit up counter (for n threads) would suffice. Figure 2.6 shows an example execution sequence from a 5 stage thread-interleaved pipeline with 5 threads. The thread-interleaved pipelines shown and presented in this thesis are all of single width. The same code segments from figure 2.3 and figure 2.1 are being executed in this pipeline. Threads 0, 2 and 4 execute GCD (figure 2.3) and threads 1 and 3 execute the data dependent code segment (figure 2.1). Each hardware thread executes as an independent context and their progress is shown in figure 2.6 with thick arrows pointing to the execution location of each thread at t0. We can observe from the figure that each time step an instruction from a different hardware thread is fetched into execution and the hardware threads are fetched in a round robin order. At time step 4 we begin to visually see that each time step, each pipeline stage is occupied by a different hardware thread. The fine-grained thread interleaving and the round robin scheduling combine to form this important property of thread-interleaved pipelines, which provides the basis for a timing predictable architecture design.

For thread-interleaved pipelines, if there are enough thread contexts, for example – the same number of threads as there are pipeline stages, then at each time step no dependency exists between the pipeline stages since they are each executing on a different thread. As a result, data and control pipeline hazards, the results of dependencies between stages within the pipelines, no longer exist in the thread-interleaved pipeline. We've already shown from figure 2.4 that when executing the GCD code segment on a single-threaded pipeline, control hazards stem from branch instructions because of the address calculation for the instruction after the branch. However, in a thread-interleaved pipeline, the instruction after the branch from the same thread is not fetched into the pipeline until the branch instruction is committed. Before that time, instructions from other threads are fetched so the pipeline is not stalled, but simply executing other thread contexts. This can be seen in figure 2.6 for thread 0, which is represented with instructions with white backgrounds. The *cmp* instructions, which determines whether next conditional branch *beq* is taken or not, completes before the *beq* is fetched at time step 5. The *blt* instruction from thread 0, fetched at time step 10, also causes no hazard because the *beq* is completed before *blt* is fetched. The code in figure 2.1 is executed on thread 1 of the thread interleave pipeline in figure 2.6. The pipeline stalls inserted from top of figure 2.2 are no longer needed even without a forwarding circuitry because the data-dependent instructions are fetched after the completion of its previous instruction. In fact, no instruction in the pipeline is dependent on another because each pipeline stage is executing on a separate hardware thread context. Therefore, the pipeline does not need to include any extra logic or hardware for handling data and control hazards in the pipeline. This gives thread-interleaved pipelines the advantage of a simpler pipeline design that requires less hardware logic, which in

turns allows the pipeline clock speed to increase. Thread-interleaved pipelines can be clocked at higher speeds since each pipeline stage contains significantly less logic needed to handle hazards. The registers and processor states use much more compact memory cells compared to the logic and muxes used to select and handle hazards, so the size footprint of thread-interleaved pipelines are also typically smaller.

For operations that have long latencies, such as memory operations or floating point operations, thread-interleaved pipelines hides the latency with its execution of other threads. Thread 3 in figure 2.6 shows the execution of a *ld* instruction that takes the same 5 cycles as shown in figure 2.2. We again assume that this *ld* instruction accesses data from the main memory. While the *ld* instruction is waiting for memory access to complete, the thread-interleaved pipeline executes instructions from other threads. The next instruction from thread 3 that is fetched into the pipeline is again the same *ld* instruction. As memory completes its execution during the execution of instructions from other threads, we replay the same instruction to pick up the results from memory and write it into registers to complete the execution of the *ld* instruction. It is possible to directly write the results back into the register file when the memory operation completes, without cycling the same instruction to pick up the results. This would require hardware additions to support and manage multiple write-back paths in the pipeline, and a multi write ported register file, so contention can be avoided with the existing executing threads. In our design we simply replay the instruction for write-backs to simplify design and piggy back on the existing write-back datapath. Multithreaded pipelines typically mark threads inactive when they are waiting for long latency operations. Inactive threads are not fetched into the pipeline, since they cannot make progress even if they are scheduled. This allows the processor to maximize throughput by allowing other threads to utilize the idle processor cycles. However, doing so has non-trivial effects on thread-interleaved pipelines and the timing of other threads.

First, if the number of "active" threads falls below the number of pipeline stages, then pipeline hazards are reintroduced; it is now possible for the pipeline to be executing two instructions from the same thread that depend on each other simultaneously. This can be circumvented by inserting pipeline bubbles when there aren't enough active threads. For example, as shown in figure 2.7, for our 5 stage thread-interleaved pipeline that has 5 threads, if two threads are waiting for main memory access and are marked inactive, then we insert 2 NOPs every round of the round-robin schedule to ensure that no two instructions from the same thread exists in the pipeline. Note that if the 5 stage thread-interleaved pipeline contained 7 threads, then even if 2 threads are waiting for memory, no NOP insertion would be needed since instructions in each pipeline stage in one cycle would still be from a different thread. NOP insertions only need to occur when the number of active threads drops below the number of pipeline stages.

The more problematic issue with setting threads inactive whenever long latency operations occur is the effect on the execution frequencies of other threads in the pipeline. When threads are scheduled and unscheduled dynamically, the other threads in the pipeline would dynamically execute more or less frequently depending on how many threads are active. This complicates timing analysis since the thread frequency of one thread now depends on the program state of all other threads. In order for multithreaded architectures to achieve predictable performance, *temporal isolation* must exist in the hardware between the threads. Temporal isolation is the isolation of timing behaviors of a thread from other thread contexts in the architecture. With temporal isolation, the timing analysis is greatly simplified, as software running on individual threads can be analyzed

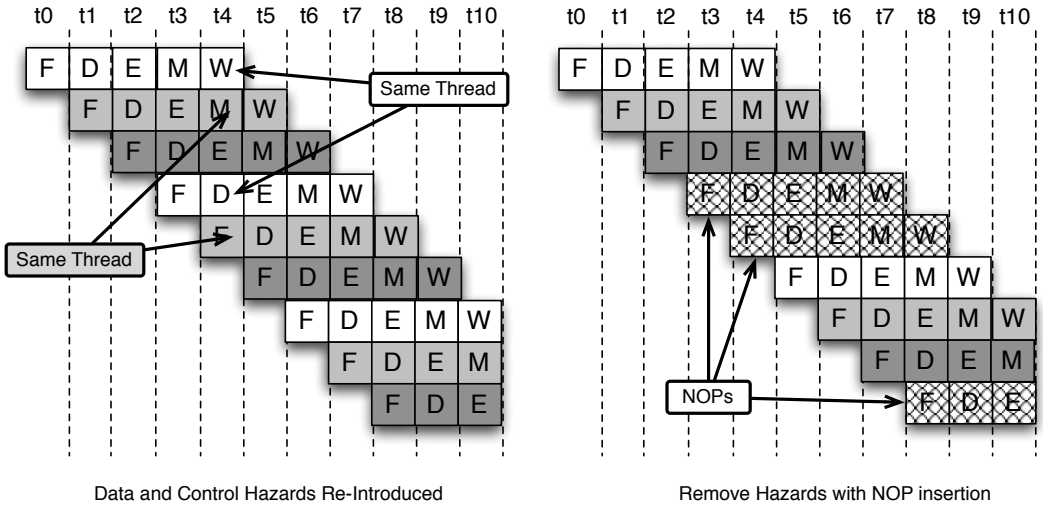


Figure 2.7: Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive

separately without worry about the effects of integration. If temporal isolation is broken, any timing analysis needs to model and explore all possible combinations of program state of all threads, which is typically infeasible. The round-robin thread scheduling of thread-interleaved pipelines is a way of achieving temporal isolation for a multithreaded architecture. Unlike coarse-grain dynamically switched multithreaded architectures, thread-interleaved pipelines can maintain the same round-robin thread schedule despite the execution context of each thread within the pipeline. This is a step towards achieving temporal isolation amongst the threads, as the execution frequency of threads does not change dynamically. However, dynamically scheduling and unscheduling threads based upon long-latency operations breaks temporal isolation amongst threads. Thus, our thread-interleaved pipeline does not mark threads inactive on long latency operations, but simply replays the instruction whenever the thread is fetched. Although this slightly reduces the utilization of the thread-interleaved pipeline, but threads are decoupled and timing analysis can be done individually for each thread without interference from other threads. At the same time, we still preserve most of the benefits of latency hiding, as other threads are still executing during the long latency operation.

(Todo: talk about xmos handling exceptions and our handling of exceptions here)

Shared hardware units within multithreaded architectures could also easily break temporal isolation amongst the threads. Two main issues arise when a hardware unit is shared between the threads. The first issue arises when shared hardware units share the same state between all threads. If the state of hardware unit is shared and can be modified by any thread, then it is nearly impossible to get a consistent view of the hardware state from a single thread during timing analysis. Shared branch predictors and caches are prime examples of how a shared hardware state can cause timing inference between threads. If a multithreaded architecture shares a branch predictor for all threads, then the branch table entries can be overwritten by branches from any thread. This means that each thread's branches can cause a branch mispredict for any other thread. Caches are especially troublesome when shared between threads in a multithreaded architecture. Not only does it make the execution time analysis substantially more difficult, it also decreases overall performance for each thread due to cache thrashing, an event where threads continuously evict each other threads cache lines in the cache(Todo: citation). To achieve temporal isolation between the threads, the

hardware units in the architecture must not share state between the threads. Each thread must have its own consistent view of the hardware unit states, without the interference from other threads. For example, each thread in our thread-interleaved pipeline contains its own private copy of the registers and thread states. We already showed why thread-interleaved pipelines do not need branch predictors because they remove control-hazards, and we will discuss a timing predictable memory hierarchy that uses scratchpads instead of caches in section 2.2. The sharing of hardware state between threads also increases security risks in multithreaded architectures. Side-channel attacks on encryption algorithms(Todo: cite) take advantage of the shared hardware states to disrupt and probe the execution time of threads running the encryption algorithm to crack the encryption key. We will discuss this in detail in section 4.2 and show how a predictable architecture can prevent timing side-channel attacks for encryption algorithms.

The second issue that arises is that shared hardware units create structural hazards – hazards that occur when a hardware unit needs to be used by two or more instructions at the same time. Structural hazards typically occur in thread-interleaved pipelines when the shared units take longer than one cycle to access. The ALU, for example, is shared between the threads. But because it takes only one cycle to access, there is no contention even when instructions continuously access the ALU in subsequent cycles. On the other hand, a floating point hardware unit typically takes several cycles to complete its computation. If two or more threads issue a floating point instruction in subsequent cycles, then contention arises, and the the second request must be queued up until the first request completes its floating point computation. This creates timing interference between the threads, because the execution time of a floating point instruction from a particular thread now depends on if other threads are also issuing floating point instructions simultaneously. If the hardware unit can be pipelined to accept inputs every processor cycle, then we can remove the the contention caused by the hardware unit, since accesses no longer need to be queued up. The shared memory system in a thread-interleaved pipeline also creates structural-hazards in the pipeline. In section 2.2 we will discuss and present our memory hierarchy along with a redesigned DRAM memory controller that supports pipelined memory accesses. If pipelining cannot be achieved, then any timing analysis of that instruction must include a conservative estimation that accounts for thread access interference and contention management. Several trade-offs need to be considered when deciding how to manage the thread contention to the hardware unit.

A time division multiplex access (TDMA) schedule to the hardware unit can be enforced to decouple the access time of threads remove timing interference. A TDMA access scheme certainly creates an non-substantial overhead compared to conventional queuing schemes, especially if access to the hardware unit is rare and sparse. However, in a TDMA scheme, each threads wait time to access the shared resource depends on the time offset in regards to the TDMA schedule, and is decoupled from the accesses of other threads. Because of that, it is possible obtain a tighter worst case execution time analysis per thread. For a TDMA scheme, the worst case access time occurs when an access just missed its time slot and must wait a full cycle before accessing the hardware unit. For a conventional queuing scheme where each requester can only have one outstanding request, the worst case happens when every other requester has a request in queue, and the first request is just beginning to be serviced. At first, it may seem that the worst case execution time of a TDMA scheme may seem similar to the basic queuing scheme. For timing analysis at an unknown state of the program, no assumption can be made on the TDMA schedule, thus the worst case time must be used for conservative estimations. However, because the TDMA access schedule is static, and

access time is decoupled from other threads, there is potential to obtain tighter timing analysis for accesses by inferring access slot hits and misses for future accesses. For example, based upon the execution time offsets of a sequence of accesses to the shared resource, we may be able to conclude that at least one access will hit its TDMA access slot and get access right away. We can also possibly derive more accurate wait times for the accesses that do not hit its access slots based upon the elapsed time between accesses. An in depth study of WCET analysis of TDMA access schedules is beyond the scope of the thesis. But these are possibilities now because there is no timing interference between the threads. A queue based mechanism would not be able to achieve better execution time analysis without taking into account the execution context of all other threads in the pipeline.

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

In this section we introduced a predictable thread-interleaved pipeline design that provides temporal isolation for all threads in the architecture. The thread-interleaved pipeline favors throughput over single thread latency, as multiple threads are executed on the pipeline in a round robin fashion. We will present in detail our implementation of this thread-interleaved pipeline in chapter 3, and show how the design decisions discussed in this chapter are applied.

2.2 Memory System

While pipelines designs continue to improve, memory technology has been struggling to keep up with the increase in clock speed and performance. Even though memory bandwidth can be improved with more bank parallelization, the memory latency remains the bottle neck to really improving memory performance. Common memory technologies used in embedded systems contain a significant trade off between access latency and capacity. Static Random-Access Memories (SRAM) provides sufficient latency that allows single cycle memory access latencies from the pipeline. However, the hardware cost to implement each memory cell prohibits large capacities to be implemented close to the processor. On the other hand, Dynamic Random-Access Memories (DRAM) uses a compact memory cell design that can easily be designed into larger capacity memory blocks. But the memory cell of DRAMs must be constantly refreshed due to charge leakage, and the large capacity of DRAM cells in a memory block design 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, while less access values

are placed farther away.

2.2.1 Memory Hierarchy

Caches

A *CPU Cache* (or cache) is commonly used in the memory hierarchy to manage the smaller fast access memory made of SRAMs. The cache manages of contents of the fast access memory in hardware by leveraging the spatial and temporal locality of data accesses. The main benefits of the cache is that it abstracts away the memory hierarchy from the programmer. When a cache is used, all memory accesses is routed through the cache. If the data from the memory access is on the cache, then a cache hit occurs, and the data is returned right away. However, if data is not on the cache, then a cache miss occurs, and the cache controller fetches the data from the larger memory and adjusts the memory contents on the cache. The replacement policy of the cache is used to determine which cache line, the unit of memory replacement on caches, to replace. A variety of cache replacement policies have been researched (Todo: cite) and used to optimize for memory access patterns of different applications. In fact, modern memory hierarchies often contain multiple layers of hierarchy to balance the trade-off between speed and capacity. A commonly used memory hierarchy is shown in figure 2.8. If the data value is not found in the L1 cache, then it is searched for in the L2 cache. If the L2 cache also misses, then the data is retrieved from main memory, and sent back to the CPU while the L1 and L2 cache updates its contents. Often times different replacement policies are used at different levels of the memory hierarchy to optimize the hit rate or miss latency of the memory access. Benchmarks have show that caches can have hit rates up to (Todo: find number)% (Todo: and cite).

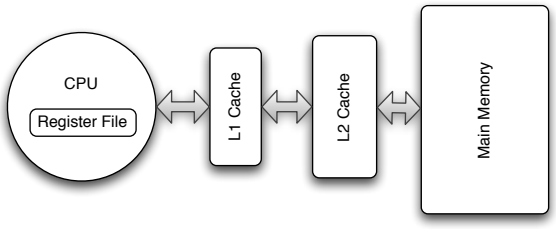


Figure 2.8: Memory Hierarchy w/ Caches

As sophisticated as this seems, the program is oblivious to the different levels of memory hierarchy, and whether or not an access hits the cache or goes all the way out to main memory. The memory hierarchy is abstracted away from the program, and the cache gives its best-effort to optimize memory access latencies. This is one of the main reasons for the cache's popularity; the programmer does not need to put in extra effort to get a reasonable amount of performance. A program can be run on any memory hierarchy configuration without modification and still obtain reasonable performance from the hardware. Thus, for general purpose applications, caches give the ability to improve design times and decrease design effort. 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, the execution context previously running, interrupt routines, and even branch mispredictions that cause unnecessary cache line replacements. Thus, when execution time is important, the variability and uncontrollability of caches may outweigh the benefits it provides.

The cache uses internal states stored in hardware to manage the replacement of contents on it. As the programmer cannot control the states of the cache explicitly, it is extremely difficult to analyze the execution time of a program running with caches. At an arbitrary point in the program,

the state of the cache is unknown to the software. Whether a memory access hits or misses the cache cannot to determined easily, so the conservative worst-case execution time analysis well need to assume the worst case as if the memory access was directly to main memory. In theory, (Todo: Cite and summarize Jan's thesis on cache analysis) showed that for certain cache replacement policies, it is possible to obtain tighter execution time analysis. However, the complexity of modern memory hierarchies with caches make timing analysis extremely difficult, and introduce high variability in program execution time.

Even outside of real-time applications, caches present side effects that it were not intended for. For applications that require extremely high speed, the best-effort memory management that caches offer simply is not good enough. In those situations, programs need to be hand tuned and tailored to specific cache architectures and parameters. Algorithm designers tune the performance of algorithms by reaching beneath the abstracted away memory architecture to enforce data access patterns to conform to the cache replacement policies and cache line sizes. Blocking [48], for example, is a well-known technique to optimize algorithms for high performance. 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. (Todo: talk about LAPACK? Libraries that tune programs to caching). In this case, we see that the hidden memory hierarchy actually could degrade program performance. Multithreaded threaded architectures with shared caches amongst the hardware threads can suffer from *cache thrashing*, an effect where different threads' memory accesses evict the cached lines of others. In this situation, it is impossible for hardware 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 keeping data 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(Todo: cite) or implementing a directory protocol(Todo: cite) have been proposed to keep the data consistent in all caches. However, doing this scalably and efficiently is still a hot topic of research today(Todo: cite all work on cache coherency).

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 from the programmer, we propose to expose the memory layout to the software. *Scratchpads* were initially proposed for their power saving benefits over caches(Todo: cite). (Todo: Papers that show scratchpads were already popular? Mention that they are already used. The Cell processor, Nvidia's 8800 GPU) Scratchpads uses the same memory technology as caches, but does not implement the hardware controller to manage its memory contents. Scratchpads have reduced access latency, area and power consumption compared to caches. Memory operations that access the scratchpad region take only a single cycle to complete, which is the same as a cache hit. Thus, scratchpads may serve as the fast-access memory in a memory hierarchy, instead of caches. Unlike caches that overlay address space with main memory, scratchpads occupy a distinct address space in memory, so it does

not need to check whether the data is on the scratchpad or not. Furthermore, the memory access time of each memory request is only depend on the memory address it is accessing. This drastically improves the predictability of memory access times, and reduces the variability of execution time introduced with caches. By using scratchpads, we explicitly expose the memory hierarchy to software, giving the programmer full control over the management of memory contents within the memory hierarchy. (Todo: show figure of memory hierarchy with scratchpads)

Two allocation schemes are commonly employed to manage the contents of scratchpads in software. Static allocation schemes allocate data on the scratchpad during compile time, and the contents allocated on the scratchpad does not change throughout program execution. Static scratchpad allocation schemes [84, 65] often use heuristics or compiler-based static analysis(Todo: citation) of program to find the most commonly executed instructions or data structures, and allocate them statically on the scratchpad to improve program performance. Dynamic allocation schemes modify the data on the scratchpad during run time in software through DMA mechanisms. The allocation could either be automatically generated and inserted by the compiler, or explicitly specified by the user programmatically. Embedded system designs typically deal with limited resources and other design constraints, such as less memory or hard timing deadlines. Thus, the design of these systems often contain analysis on memory usage etc to ensure that the constraints are met. Actor oriented models of computations, such as Dataflow(Todo: cite) or Giotto(Todo: cite), allow users to design systems at a higher level. These higher level actor oriented programming models exposes the structure and semantics of the model for better analysis, which can be used to optimize scratchpad allocation dynamically. Bandyopadhyay [12] presented an automated memory allocation of scratchpads for the execution of Heterochronous Dataflow models. The Heterochronous Dataflow (HDF) model is an extension to the Synchronous Dataflow (SDF) model with finite state machines (FCM). The HDF models contain different program states, each state executing a SDF model that contains actors communicating with each other. Bandyopadhyay analyzed the actor code and the data that was being communicated in each HDF state. The dynamic scratchpad allocation is inserted during state transitions, and the memory allocated is optimized for each HDF state. This allocation not only showed roughly 17% performance improvement compared to executions using LRU caches, but also more predictable program performance.

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

2.2.2 DRAM Memory Controller

Because of its high capacity, DRAMs are often employed in modern embedded systems to cope with the increasing code and data sizes. However, bank conflicts and refreshes within the DRAM can cause memory accesses to stall, further increasing the memory latency. Modern memory controllers are designed to optimize average-case performance by queueing and reordering memory requests to improve throughput of memory requests. This results in unpredictable and varying

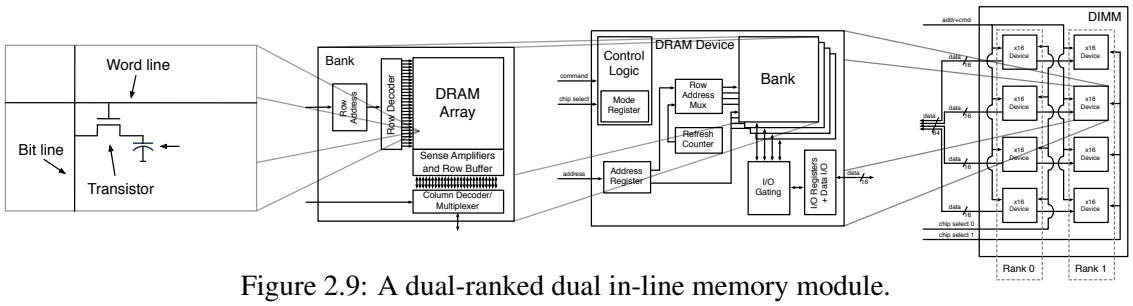


Figure 2.9: A dual-ranked dual in-line memory module.

access times and increased worst-case access time for each memory request. In this section we will present a DRAM memory controller that privatizes DRAM banks with scheduled memory refreshes to provide improved worst-case latency and predictable access time. The contributions from this section is research done jointly with several co-authors from Reineke et. al [74]. We do not claim sole credit for this work, and the summary is included in this thesis only for completeness purposes. We will first give some basic background on DRAM memories, then present the predictable DRAM controller designed.

DRAM Basics

Figure 2.9 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 [39]. A ***DRAM array*** is made of a two-dimensional array of DRAM cells. Each access is 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 closed to the voltage threshold so the sense amplifiers can detect the bit value. Columns can be read and written to quickly after the row is in the row buffer. The ***DRAM device*** consists of banks formed of DRAM arrays. Modern DRAM devices have multiple banks, control logic, and I/O mechanisms to read from and write to the data bus as shown in the center of 2.9. Banks can be accessed concurrently, but the data, command and address busses are shared within the device, which is what the memory controller uses to send commands to the DRAM device. The following table[1] lists the four most important commands and their function:

[1] This table is as shown in [74]

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

To perform reads or writes, the controller first sends the PRE command to precharge the bank containing the data. Then, a RAS is issued to select the row, and one or more CAS commands can be used to access the columns within the row. Accessing columns from the same row does not require additional PRE and RAS commands, thus higher throughput can be achieved by performing column accesses in burst lengths of four to eight words. Column accesses can immediately be followed by a PRE command to decrease latency when accessing different rows. This is known as auto-precharge (or closed-page policy). Refreshing of the cells can be done in two ways. First, by issuing a refresh command, which refreshes all banks of the device simultaneously. The refresh latency depends on the capacity of the device, but the DRAM device manages a counter to step through all the rows. The rows on the device could also be manually refreshed by performing row accesses to them. Thus, the memory controller could performance row accesses on every row within the 64 ms refresh period. This requires the memory controller to keep track of the refresh status of the device and issue more refresh commands, but each refresh takes less time because it is only a row access. ***DRAM modules*** are made of several DRAM devices integrated together for higher bandwidth and capacity. A high-level view of the dual-ranked dual in-line memory module (DIMM) is shown in the right side of 2.9. The DIMM has eight DRAM devices that are organized in two ranks, which share the address, command and data bus. The two ranks share the address, command inputs, and the 64-bit data bus. The chip is used to determine which ranks is addressed. All devices within a rank are accessed simultaneously when the rank is addressed, and the results are combined to form the request response. Our controller makes use of a feature from the DDR2 standard known as posted-CAS. Unlike DDR or other previous versions of DRAMs, DDR2 can delay the execution of CAS commands (posted-CAS) for a user-defined latency, known as the additive latency (AL). Posted-CAS can be used to resolve command bus contention by sending the posted-CAS earlier than the corresponding CAS needs to be executed.

Table 2.1 gives an overview of timing parameters for a DDR2-400 memory module. These timing constraints come from the internal structure of DRAM modules and DRAM cells. For example, t_{RCD}, t_{RP}, and t_{RFC} are from the structure of DRAM banks that are accessed through sense amplifiers that need to be precharged. t_{CL}, t_{WR}, t_{WTR}, and t_{WL} result from the structure of DRAM banks and DRAM devices. The four-bank activation window constraint t_{FAW} constrains rapid activation of multiple banks which would result in too high a current draw. The memory controller must conform to these timing constraints when sending commands to the DDR2 module. Here we only gave a quick overview of DRAMs, we refer more interested readers to Jacob et al. [38] for more details.

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

Parameter	Value (in cycles at 200 MHz)	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	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. [74]

Predictable DRAM Controller

We will split the discussion of the predictable DRAM controller into its backend and frontend. The backend translates memory requests into DRAM commands that are sent to the DRAM module. The frontend manages the interface to the pipeline along with the responsibility of scheduling the refreshes. Here we specifically refer to a DDR2 667MHz/PC2-5300 memory module operating at 200Mhz, which has a total size of 512MB over two ranks with four banks on each rank. While our discussion of the design of this DRAM controller is specific to our DDR2 memory module, the key design features are applicable to other modern memory modules.

Backend Conventional DRAM memory controllers view the entire memory device as one resource, and any memory requests can access the whole DRAM device. Subsequent memory accesses can target the same bank within the DRAM, which results in the need for memory requests to be queued and serviced sequentially, without exploiting bank parallelism. Our controller views the memory devices as independent resource partitioned by banks. Specifically, we partition our memory module into four resources, each consisting of two banks within the same rank. The banks within each partition 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 sharing of resources 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 different resources grouped from banks. Figure 2.10 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.

Each access request is translated into a RAS (Row Access), posted-CAS (Column Access) and NOP command, which we call an access slot. The NOP command in the access slot is inserted

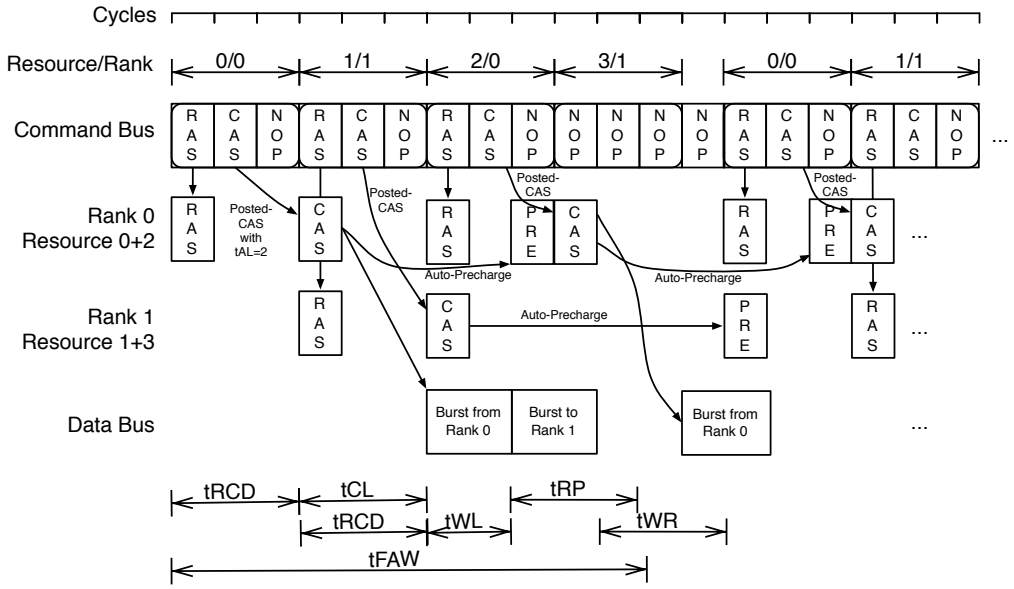


Figure 2.10: The periodic and pipelined access scheme employed by the backend [74].

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 2.1. This latency specifies that the RAS command and the first CAS command need to be 3 cycles apart. However, figure 2.10 shows that manually issuing a CAS command to the first resource 3 cycles after its RAS command would cause a command bus conflict with the RAS command for the second resource. Thus, we instead set the additive latency t_{AL} to 2 and use the posted-CAS that offsets the CAS command to conform to the row to column latency. This allows our memory controller to preserve our pipelined access scheme while meeting the latency requirements of the DRAM. We use a closed-page policy (also known as auto-precharge policy), which causes the accessed row to be immediately precharged after performing the column access (CAS), preparing it for the next row access. If there are no requests for a resource, the backend does not send any commands to the memory module, as is the case for resource 3 in 2.10.

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

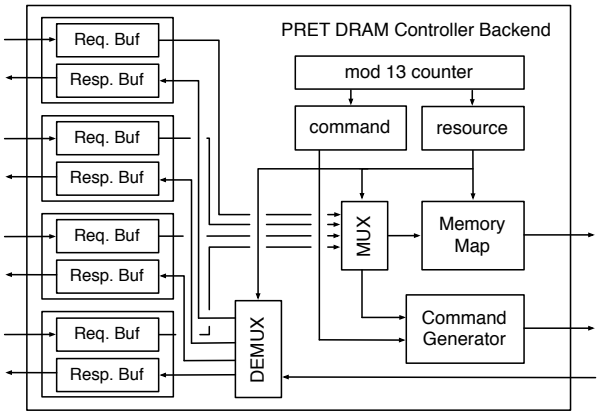


Figure 2.11: Sketch of implementation of the backend [74].

Frontend The frontend of our memory controller manages the interfacing to our backend, and the refreshing of the DRAM device. The privatization of DRAM banks creates four independent resources that is to be accessed separately from the front end. Thus, our memory controller is designed to be used by multicore or multithreaded architectures that contain multiple requesters which need access to the main memory. Several recent projects strive to develop predictable multi-core architectures, such as those proposed by the MERASA [89], PREDATOR [98], JOP [81], or CoMP-SoC [34] projects, which require predictable and composable memory performance. These could potentially profit from using the proposed DRAM controller. Specifically, we designed this memory controller to interface with the thread-interleaved pipeline discussed previously in section 2.1.3. The thread-interleaved pipeline contains multiple hardware threads that each require access to main memory independently. 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.

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, we dedicate a region on the shared scratchpad that all the threads on our thread-interleaved pipeline can access in a single processor cycle. This can be done because the scratchpad and DRAM memory has distinct address regions, so no shared memory space will overlap onto the DRAM. However, most multi-core processors use DRAM to share data while local scratchpads or caches are private. The sharing of data on the DRAM can be acheived by arbitrating accesses in the frontend. In this case, the four resources in the backend can be combined into one, and any access to this single resource would result in four smaller accesses to all the backend resources. This single resource could then be shared among the different cores of a multi-core architecture using predictable arbitration mechanisms such as Round-Robin or CCSP [7] 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 [74].

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[3]. Hardware refresh commands have longer refresh latencies each time a refresh is issued, but requires less refresh commands to meet the refresh constraints posed by the DRAM. However, when the hardware refresh command is issued, all banks in the target DRAM device are refreshed, prohibiting any other memory access to the device. In our backend, this would extend across multiple resources, causing multiple resources to be blocked for memory accesses. Memory access latencies now need to account for potential refresh command latencies, which varies depending on the refresh progress. Instead, we use the distributed, RAS-only refresh [56] to each bank separately. Memory refreshes is in this case is equivalent to a row accesses to a bank, and each resource can be refreshed without effecting others. Manually accessing rows on the other have much shorter latencies each time, but incurs a slight bandwidth hit because more accesses need to be performed to meet the refresh constraints. The shorter latencies however improves the worst-case access latency, because the refresh latency is shorter.

When a refresh is required can be statically analyzed. In our device, each bank consists of 8192 rows, so each row has to be refreshed every $64ms/8192 = 7.8125\mu s$. At a clock rate of 200 MHz of the memory controller, this corresponds to $7.8125\mu s \cdot (200cycles/\mu s) = 1562.5$ cycles. Since each resource contains two banks, we need to perform two refreshes every 1562.5 cycles, or one every 781.25 cycles. One round of access is 13 cycles at burst length 4, and includes the access slots to each resource plus a nop command. So in the frontend we schedule a refresh every $\lfloor 781.25/13 \rfloor^{th} = 60^{th}$ round of the backend. If no memory access is in the request buffer for the resource being scheduled for refresh, then the row refresh can be directly be issued. Typically when a contention between a memory request and a refresh occurs, the refresh gets priority so the data can be retained in the DRAM cell. However, our refresh schedule schedules refreshes slightly more often than necessary. Scheduling a refresh every $60 \cdot 13$ cycles means that every row, and thus every DRAM cell, is refreshed every $60 \cdot 13\ cycles \cdot 8192 \cdot 2/(200000\ cycles/ms) \leq 63.90ms$. We can thus push back any of these refreshes individually by up to $0.1ms = 20000$ cycles without violating the refreshing requirement. So in our frontend, the memory request is serviced first (which takes 13 cycles), then the refresh is issued in the next access slot. In section 3.2 when

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

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

2.3 Instruction Set Extensions

(Todo: also add in a description of delay and set, which is used in the 1dcfd application)

Chapter 3

Implementation of PTARM

The Precision Timed ARM (PTARM) architecture is a realization of the PRET principles on an ARM ISA architecture(Todo: Citation). In this chapter we will describe in detail the implementation details of the timing-predictable ARM processor and discuss the worst-case execution time analysis of code running on it. We show that with the architectural design principles of PRET, the PTARM architecture is easy analyzable with repeatable timing.

The architecture of PTARM closely follows the principles discussed in chapter 2. This includes a thread-interleaved pipeline with scratchpads along with the timing predictable memory controller. The ARM ISA was chosen not only for its popularity in the embedded community, but also because it is a Reduced Instruction Set Computer (RISC), which has simpler instructions that allow more precise timing analysis. Complex Instruction Set Computers (CSIC) on the other hand adds un-needed complexity to the hardware and timing analysis. RISC architectures typically features a large uniform register file, a load/store architecture, and fixed-length instructions. In addition to these, ARM also contains several unique features. ARM's ISA requires a built in hardware shifter along with the arithmetic logic unit (ALU), as all of its data-processing instructions can shift its operands before passed onto the ALU. ARM's load/store instructions also contain auto-increment capabilities that can increment or decrement the value stored in the base address register. This is useful to compact code that is reading through an array in a loop, as one instruction can load the contents and prepare for the next load in one instruction. In addition, almost all of the ARM instructions are conditionally executed. The conditional execution improves architecture throughput with potential added benefits of code compaction(Todo: Citation). ARM programmer's model specifies 16 general purpose registers (R0 to R15) to be accessed with its instructions, with register 15 being the program counter (PC). Writing to R15 triggers a branch, and reading from R15 reads the current PC plus 8.

ARM has a rich history of versions for their ISA, and PTARM implements the ARMv4 ISA, currently without support for the thumb mode. PTARM uses scratchpads instead of caches, and a DDR2 DRAM for main memory managed by the timing predictable memory controller. PTARM also implements the timing instructions introduced in chapter 2.3.

3.1 Thread-Interleaved Pipeline

PTARM implements a thread-interleaved pipeline for the ARM instruction set. PTARM was initially written to target Xilinx Virtex-5 Family FPGAs, thus several design decisions were made to optimized the PTARM architecture for Xilinx V5 FPGAs. PTARM has a 32 bit datapath in a five stage pipeline with four threads interleaving through the pipeline. Chapter 2 discussed the timing and hardware benefits of a typical thread-interleaved pipeline which removes pipeline hazards with multiple threads. Section 2.1.3 mentioned that conventional thread-interleaved pipelines typically have at least as many threads as pipeline stages to keep the pipeline design simple and maximize the clock speed. However, having more threads in the pipeline increases single thread latency, since all threads are essentially time-sharing the pipeline resource. Lee and Messerschmitt [51] showed that the minimum number of threads required to remove hazards is actually less than the number of pipeline stages in the pipeline. In our design, we implement a five stage thread-interleaved pipeline with four threads by carefully designing the PC writeback mechanism one pipeline stage earlier.

Figure 3.1 shows a block diagram view of the pipeline. Some multiplexers within the pipeline have been omitted in the figure for a simplified view of the hardware components that make up the pipeline. There contains four copies of the Program Counter(PC), Thread States, and Register File. Most of the pipeline design follows a typical Hennessy and Patterson(Todo: citation) five stage pipeline, with the five stages in the pipeline being – Fetch, Decode, Execute, Memory, Writeback. We will briefly describe the functionality of each stage, and leave more details when we discuss how instructions are implemented in section 3.4.

The *fetch stage* of the pipeline selects the correct PC according to which thread is executing, and passes the address to instruction memory. The PC forward path forwards a loaded address from main memory for instructions that load to R15, which causes a branch. We will discuss the need for the forwarding path below when we describe the *writeback stage*. A simple $log(n)$ bit upcounter is used to keep track of which thread current to fetch.

The *decode stage* contains the *pipeline controller* which does the full decoding of instructions and sets the correct pipeline signals to be propagated down the pipeline. Most of ARM instructions are conditionally executed, so the pipeline controller first checks the condition bis to determine whether the instruction is to be executed or not. Typically the *pipeline controller* needs to

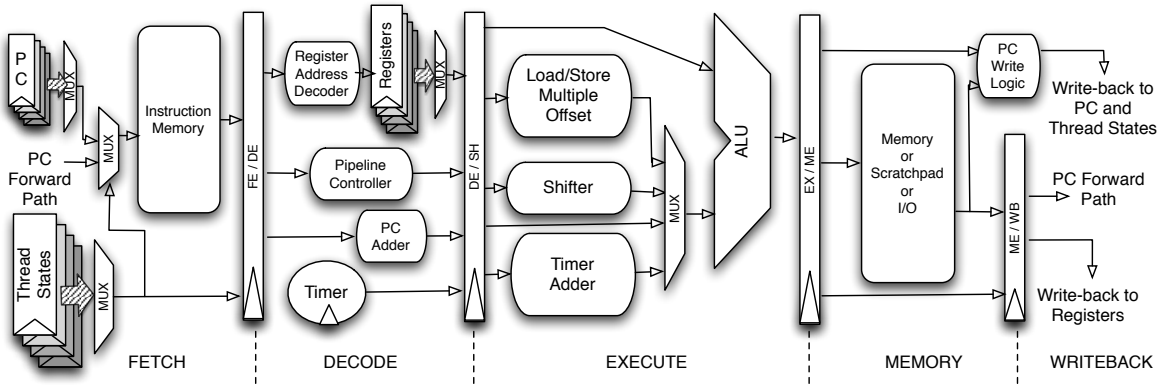


Figure 3.1: Block Level View of the PTARM 5 stage pipeline

know the current instructions in the pipeline to detect the possibility of pipeline hazards and stall the current instruction. However,in a thread-interleaved pipeline, other instructions down the pipeline are from threads, thus the controller logic is greatly simplified. It simply decodes the instruction to determine the correct signals to send to the data-path and multiplexers down the pipeline. It does not need to know any information about instructions already in flight. A small decoding logic, the *register address decoder*, is inserted in parallel with the controller to decode the register addresses from the instruction bits. Typical RISC instruction sets, such as MIPS, set the encoding of instruction bits so the register operands have a fixed location for all instruction types. However, in the ARM instruction set, certain instructions encode the register read address at different bit locations of the instruction. For example, ARM data-processing register shift instructions reads a third operand from the register to determine the shift amount. Store instructions also read a third register to obtain the register value that is stored to memory. However, both instructions have different bit locations in the instruction encoding to determine what register to read from. 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 used to increment the PC. The ARM ISA programmer's model states that reading from R15 reads the current PC+8, the PC adder not only increments the PC by 4 to get the potential next PC, but it also increments the current PC by 8 to be used as an operand. Single threaded pipelines need to increment the PC immediately in the fetch stage to prepare for the next instruction fetch. For thread-interleaved pipelines, since the next PC from the current thread is not needed until several cycles later, it doesn't need to be in the fetch stage. But because we need the results of PC+8 as a data operand, it is placed in the decode stage. The *timer* is a hardware counter clocked to the processor clock which is used to implement the timing instructions mentioned in section 2.3. The timer contains a 64 bit value that represents nanoseconds, and starts at 0 when the pipeline starts up. The time value is latched in the decode stage as the subsequent stages use it for timer manipulation.

The *execute*, *memory* and *writeback* stages execute the instruction and commits the result. The *execute* stage contains mostly execution units and muxes that select the correct operand and feeds it to the ALU. The ARM ISA assumes a built in shifter to shift the operands before operations, so a 32 bit *shifter* is included to shift the operands before the ALU. The *load/store multiple offset* logic block is used to calculate the offset of load/store multiple instructions. The load/store multiple instruction uses a 16 bit vector to represent each of the 16 general purpose registers. The bits that are set in that bit vector represents a load/store on that register. The an offset is added to the base memory address for the instruction, and that offset depends on how many bits are set. Thus, the load/store multiple offset logic block does a bit count on the bit vector and adjusts the offset to be passed into the ALU for load/store multiple instructions. The *timer adder* logic block is a 32 bit add/subtract unit. Time in the pipeline is a 64 bit value representing nanoseconds. Thus, any timing instruction that interacts with the timer in the pipeline needs to operate on 64 bit values. We could have reuse the existing ALU at the expense of having all timing instructions take an additional pass through the pipeline. But we chose to include an addition add/subtract unit specifically for the implementation of the delay_until instruction so it can check for deadline expiration every cycle, which we will discuss in detail in section 3.4 when we show how delay_until is implemented. A 32 bit *ALU* does most of the logical and arithmetic operations, including data-processing operations and branch address calculations. The results is passed to the *memory stage*, which either uses it as an address to interact with the data memory, or forwards it along to the *writeback stage* to commit back to the registers.

Figure 3.2 shows an execution sequence of the four thread five stage pipeline. The instruction in the fetch stage belongs to the same thread as the instruction in the writeback stage. This does not cause any data hazards because the data from the registers will not be read until the decode stage. But committing the PC at the writeback stage would result in a control hazard because the PC would not be ready for the subsequent fetch. For most instructions, the next PC calculation is completed before the memory stage, so we move the PC commit one stage earlier so the next instruction can be fetched. However, the ARM ISA allows instructions to write to register 15 (PC), which acts as a branch to the value written to R15. This means a load instruction can write to R15 and cause a branch whose target is not known until after the memory read. Thus, a PC forwarding path is added to forward the PC back from memory if a load instruction writes to R15. The forwarding path does not cause any timing analysis difficulties because the statically the forwarding path is only used when a load instruction writes to R15, which can be statically determined. Also, this causes no stall in the pipeline, and does not effect the timing of any following instructions. This allows us to interleave four threads in our five stage pipeline instead of five.

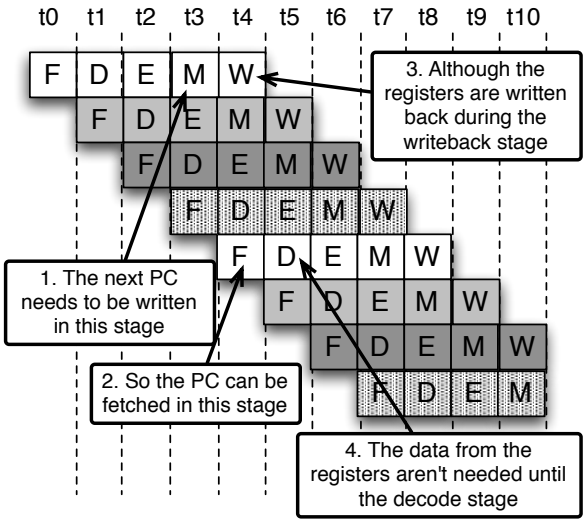


Figure 3.2: Four thread execution in PTARM

3.2 Memory Hierarchy

The memory hierarchy of PTARM is exposed to the programmer, as discussed in section 2.2. It is composed of a boot ROM (read only memory), instruction and data scratchpads, the predictable DRAM controller, and a memory mapped I/O region, all occupying separate address spaces. Figure 3.3 shows the address regions occupied by each memory type.

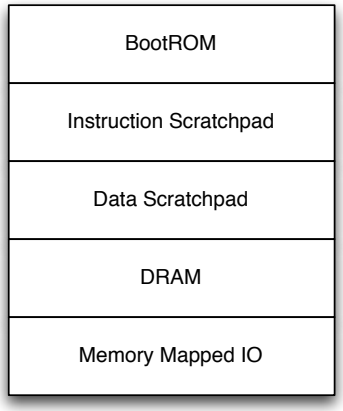


Figure 3.3: Memory Layout of PTARM

3.2.1 Boot ROM

The boot ROM (Todo: mention boot code size) contains the reset instructions and exception vector table that stores entries for handling different exceptions that occur in the pipeline. It also contains shared exception handlers and initialization code. The instructions store on the boot ROM cannot be modified at run-time, hence the read-only memory name. How-

ever, a dedicated region in the boot ROM stores a table for user-registered exception handlers, these entries can be modified prorammatically. We use this region to allow users to register timer expire exception handlers.

3.2.2 Scratchpads

The instruction and data scratchpad (Todo: size?) can be partitioned into private regions for each thread, or shared by all threads. If PTARM is used in embedded security application, such as running encryption algorithms on it, then the partitioning the scratchpads into private regions might be desired. In chapter 4 we will discuss the security implications and how a Precision Timed Machine can defend against side-channel attacks. On the other hand, sharing the scratchpad could provide flexibility on the allocation of scratchpads between hardware threads. More space could be allocated to hardware threads running more memory intensive tasks while less could be allocated to the other hardware threads (Todo: cite scratchpad allocation paper). Both the boot ROM and scratchpads are synthesized to dual-ported block RAMs on the FPGA, and provide deterministic single cycle access latencies.

3.2.3 DRAM

Access to

However, due to the thread-interleaving, only one thread can access the scratchpad at any time. Each hardware thread is also equipped with a direct memory access (DMA) unit, which can perform bulk transfers between the two scratchpads and the DRAM. Both scratchpads are dual-ported, allowing a DMA unit to access the scratchpads in the same cycles as its corresponding hardware thread. In our implementation of thread-interleaving, if one thread is stalled waiting for a memory access, the other threads are unaffected and continue to execute normally.

The four resources provided by the backend are a perfect match for the four hardware threads in the PTARM thread-interleaved pipeline. We assign exclusive access to one of the four resources to each thread. In contrast to conventional memory architectures, in which the processor interacts with DRAM only by filling and writing back cache lines, there are two ways the threads can interact with the DRAM in our design. First, threads can initiate DMA transfers to transfer bulk data to and from the scratchpad. Second, since the scratchpad and DRAM are assigned distinct memory regions, threads can also directly access the DRAM through load and store instructions.

Whenever a thread initiates a DMA transfer, it passes access to the DRAM to its DMA unit, which returns access once it has finished the transfer. During the time of the transfer, the thread can continue processing and accessing the two scratchpads. If at any point the thread tries to access the DRAM, it will be blocked until the DMA transfer has been completed. Similarly, accesses to the region of the scratchpad which are being transferred from or to will stall the hardware thread[1]. 3.4 shows a block diagram of PTARM including the PRET DRAM controller backend and the memory module. The purpose of the frontend is to route requests to the right request buffer in the backend and to insert a sufficient amount of refresh commands, which we will discuss in more detail.

When threads directly access the DRAM through load (read) and store (write) instructions, the memory requests are issued directly from the pipeline. **??**, which we will later use to

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

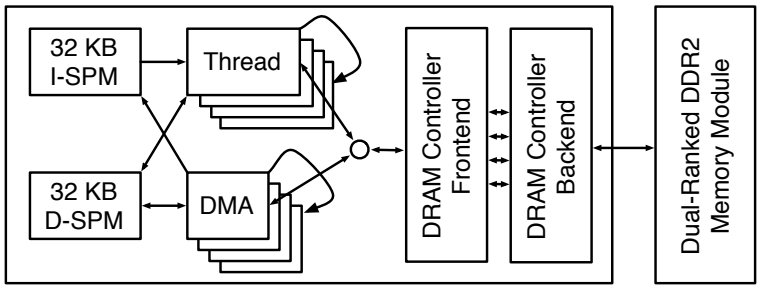


Figure 3.4: Integration of PTARM core with DMA units, PRET memory controller and dual-ranked DIMM [74].

derive the read latency, illustrates the stages of the execution of a read instruction in the pipeline. At the end of the memory stage, a request is put into the request buffer of the backend. 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 pipeline to reach the write-back stage of the corresponding hardware thread. Unlike the thread-interleaved pipeline, the DMA units are not pipelined, which implies that there are no "alignment losses": the DMA units can fully utilize the bandwidth provided by the backend.

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 which are not preceded by other stores can be performed in a single thread cycle. By *thread cycle*, we denote the time it takes for an instruction to pass through the thread-interleaved pipeline. Other stores may take two thread cycles to execute. A bigger store buffer would be able to hide latencies of successive stores at the expense of increased complexity in timing analysis.

DMA refreshes We make use of this flexibility for loads from the pipeline and when performing DMA transfers: if a load would coincide with a scheduled refresh, we push back the refresh to the next slot. Similarly, we skip the first refresh during a DMA transfer and schedule an additional one at the end of the transfer. This pushes back the refresh of a particular row by at most $60 \cdot 13$ cycles. More sophisticated schemes would be possible, however, we believe their benefit would be slim. Following this approach, two latencies can be associated with a DMA transfer:

1. The time from initiating the DMA transfer until the data has been transferred, and is, e.g., available in the data scratchpad.

2. The time from initiating the DMA transfer until the thread-interleaved pipeline regains access to the DRAM.

Our conjecture is that latency 1 is usually more important than latency 2. Furthermore, our approach does not deteriorate latency 2. For loads sent from the pipeline, the pushed back refreshes become invisible: as the pipeline is waiting for the data to be returned and takes some time to reach the memory stage of the next instruction, it is not able to use successive access slots of the backend, and thus it is unable to observe the refresh at all. With this refresh scheme, refreshes do not affect the latencies of load/store instructions, and the refreshes scheduled within DMA transfers are predictable so the latency effects of the refresh can be easily analyzed.

3.2.4 Memory Mapped I/O

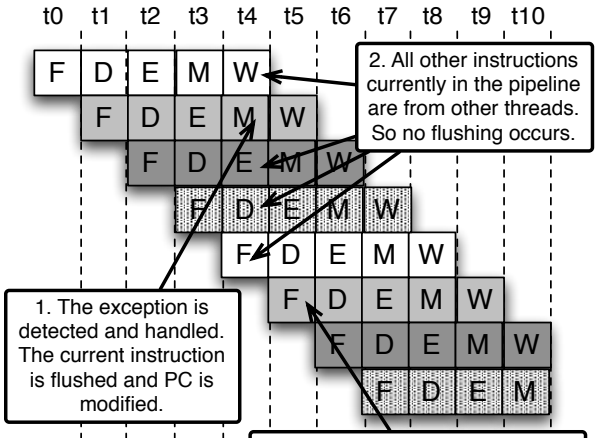
Currently PTARM implements a separate I/O bus for communicating with I/O. The bus is currently shared between all threads.

3.3 Exception Handling

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 become invalid, and the pipeline overwrites the PC to jump to a specified exception handler. The ARM ISA specifies seven types of exceptions, and an exception vector table that points the PC to specified handler addresses when those exceptions occur. The exception vector table is stored in the BootROM in our implementation. Exceptions can be triggered by external or internal events that occur in the pipeline, such as an toggling an external interrupt signal or using a software interrupt instruction to trigger the exception programmatically. But no matter how the exceptions are generated, they must be handled predictably in the pipeline.

In the context of a thread interleaved pipeline, all threads are temporally isolated. Thus, an exception that occurs on one thread must not effect the execution of other threads in the pipeline. In our pipeline, any exceptions or interrupts that occur during execution are latched at each stage and propagated down the pipeline with the instruction. An instruction flush signal is toggled to ensure that this instruction does not commit any state to memory or registers. The exception type is checked at the memory stage in the PC write logic before the next PC is committed. According to the exception type, the program state register bits are set, and the PC is redirected to the correct entry in the exception vector table. The current PC is passed on to the writeback stage to store in the link register (R14). This provides a mechanism for the program return to the initial instruction where the exception occurs and re-execute it if desired, since the instruction did not complete its execution. If the exception is generated from after a memory access and detected in the writeback stage, the PC forwarding path is used to fetch the exception vector entry for data memory exceptions. Note that it is up the each exception handler to save the register states and stack of the program.

None of the instructions that are executing in the pipeline are flushed when an exception occurs in our pipeline. As shown in figure 3.5, the instructions that are executing in other pipeline stages all belong to other threads, so no flushing of the pipeline is required because no instruction was executed



speculatively. This simplifies the timing analysis of exceptions in our pipeline, as the timing behavior of other threads in the pipeline are unaffected. For the thread which the exception occurs, the only overhead to handling the exception is that the current instruction does not complete its execution this thread cycle. The next thread cycle the pipeline will be handling the exception already, resulting in no additional stalls for the thread.

It is possible that an exception occurs during a memory access instruction that is waiting for the results from DRAM to complete. In this case, because the memory request is also sent to the DRAM controller, and possible already being serviced by the DRAM, we cannot cancel the memory request abruptly. In the case that the interrupted instruction was a load, we can simply disregard the results of the load, but if the instruction was a store, we cannot cancel the store request that is writing data to the memory. So it is up to the programmer to disable interrupts before writing to critical memory locations that require a consistent program state. By interrupting an instruction that is waiting on memory access to complete, we also potentially complicate the interaction with our DRAM controller. The DRAM controller can only service one request from each thread at a time for predictable performance(Todo: elaborate on this?). This normally is not an issue because our pipeline does not reorder instructions or speculatively execute while there are outstanding memory requests, but the pipeline waits until the request is finished before continuing execution. However, if a memory instruction is interrupted, the pipeline flushes the current instruction and continues execution of the exception handler in the Boot ROM. If at this point, the exception handler contains a memory request instruction to the DRAM, a memory request would be issued to the DRAM controller that is still servicing the previous request prior to the exception from this thread. The current memory request in this case would need to wait until the previous "canceled" memory request to complete its service by the DRAM before it can begin being serviced. This creates timing variability to the load instructions because the execution time of load instructions would be different depending on whether an exception just occurred or not. Because this situation can only occur in the exception handler, because it contains the instructions executed right after an exception, so we leave it to the compiler to ensure that the first few instructions in the exception handler code does not access the DRAM memory region. In PTARM, the compiler simply needs to ensure that the first three instructions executed from an exception handler are not instructions that access the DRAM.

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, we might have a hard real-time task that is executing on one thread, while another thread 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 effected even if it is in execution when the interrupt occurs. Only the specific thread handling the UART transfers should be interrupt by this interrupt. So we envision an interrupt controller that allows each thread to register specific interrupts that it handles, without affecting other threads in the pipeline.

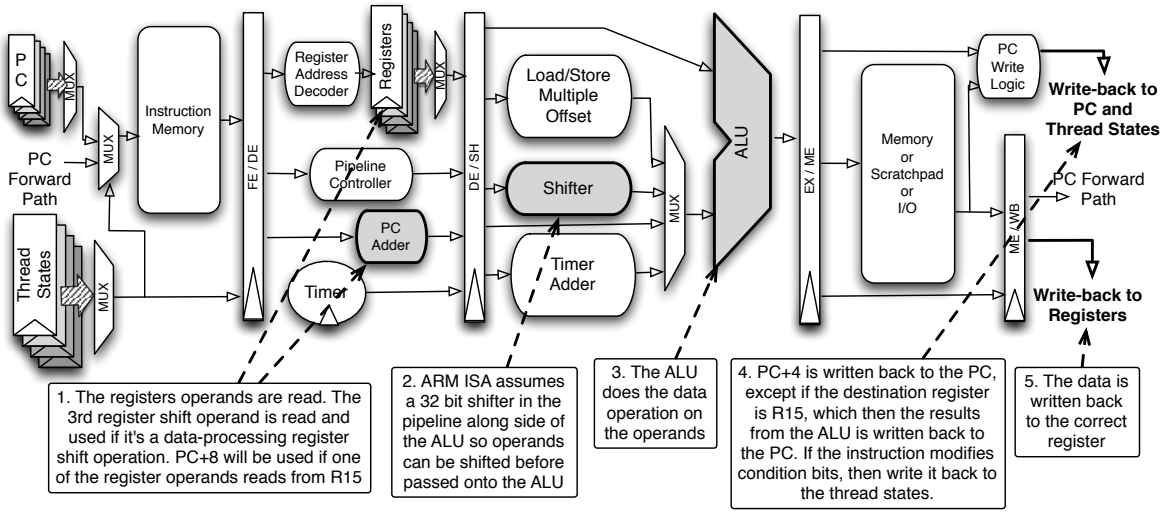


Figure 3.6: Data Processing Instruction Execution in the PTARM Pipeline

3.4 Instruction Implementations

In this section we go into more details on how each instruction type is implemented and how each hardware block in the pipeline shown in figure 3.1. We will go through different instruction types and discuss the timing implications each instruction in our implementation. We will summarize with a table with all instructions and the cycle count it takes to execute them.

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. One operand is always a register value, the second operand is labeled the shifter operand. The shifter operand could be an immediate value or a register value, both which can be shifted to form the final operand that is fed into the ALU. Figure 3.6 explains how data-processing instructions are executed through the pipeline.

Because R15 is PC, so data-processing 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 to the result of the computation. As discussed earlier, our pipeline commits the next PC in the memory stage, so to trigger a branch from data-processing instructions simply means storing back the results from the ALU as the next PC. In our thread-interleaved pipeline, when the next PC from the current thread is fetched, it will contain already contain the target address to branch to when we issue a data-processing instruction that writes to R15.

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

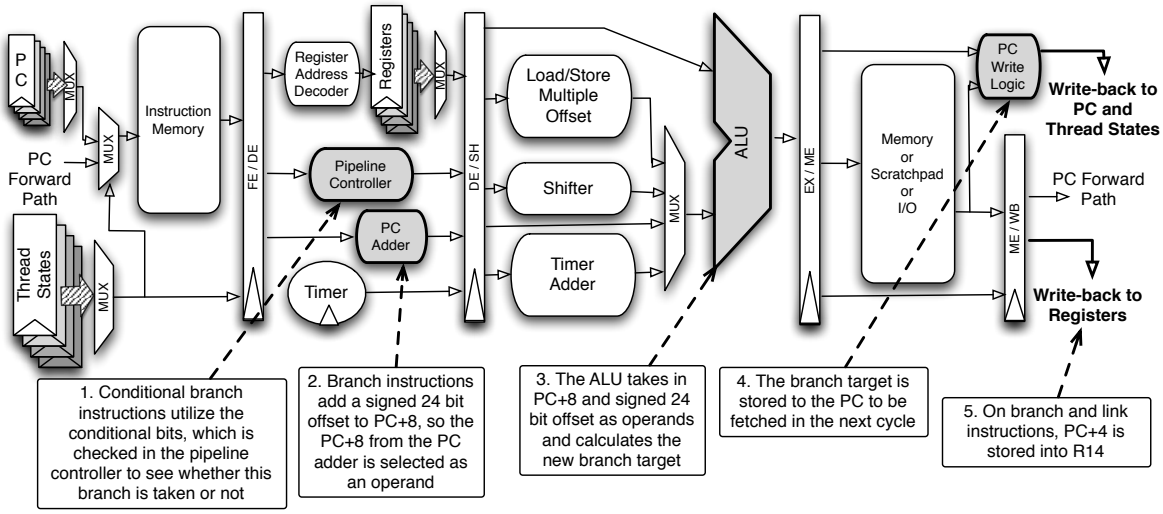


Figure 3.7: Branch Instruction Execution in the PTARM Pipeline

nism to update the condition code flags according to the results of data operations. The instructions that update the flags do not write any data back to the registers, they simply update the condition code flags.

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 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. So the condition used to determine if a conditional branch is taken is simply the condition code flags in the thread state. Figure 3.7 show how branch instructions are executed in the thread-interleaved 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 adder, in addition to incrementing the PC by the conventional offset of 4, also increments the PC by 8, to be used as an operand for the ALU to calculate the target branch address. Once the address is calculated, it is written back to its thread's next PC ready to be fetched. If the instruction is a branch and link (*bl*) instruction, PC+4 is propagated down the pipeline and written back to the link register (R14).

All branch instructions, whether conditionally taken or not, all take only one thread cycle to execute. But more importantly, the next instruction after the branch, whether it is a conditional branch or not, is not stalled or speculatively executed. The execution time of instructions from the same thread after the branch is not stalled nor affected by the branch instruction. The thread-interleaved pipeline simplified the implementation of the branch instruction and control hazard handling logic, as the pipeline will not need the results of the branch target address calculation the very next processor cycle. Instead, instructions from other threads will be fetched before the results of the branch is needed.

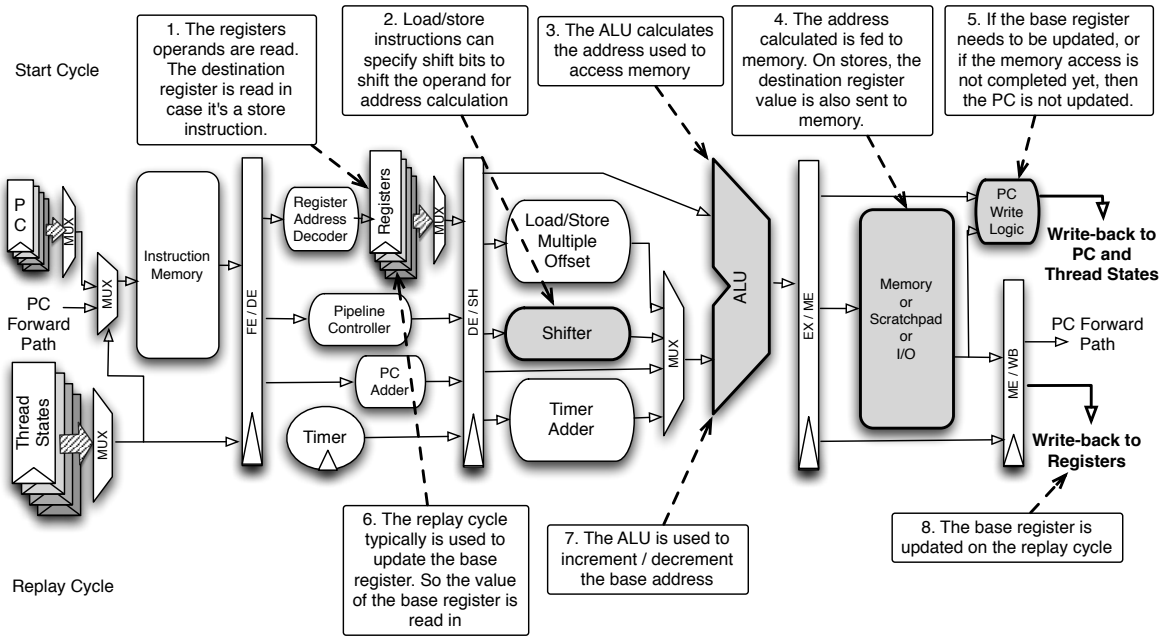


Figure 3.8: Load/Store Instruction Execution in the Ptarm Pipeline

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 in particular, the interaction of the pipeline with the memory hierarchy presented earlier. We also present a special case when the load instruction loads to R15, which loads a branch target address from memory and triggers a branch. This slightly complicates our pipeline design, but we show that it does not affect the timing and execution of the instruction and subsequent instructions. Currently load/store halfword doubleword is not implemented in PTARM, as they fall under the miscellaneous instructions category. These instructions can easily be implemented using the same principles described below without significant hardware additions.

Load/Store Register

Load instructions load data from the memory and writes them into the registers. Store instructions store data from the registers into memory, thus store instructions utilize the extra register read port to read in the register value to be stored into memory. The address used to access memory is formed by combining a base register and an offset value. The offset value can be a 12 bit immediate supplied from the instruction, or a register operand that can be shifted. The current load/store instructions can support word operations or byte operations. Figure 3.8 shows how the load/store instruction is executed in the pipeline.

All load and store instructions in ARM have the ability to update the base register after any memory operation. This compacts code that reads arrays, as a load or store instruction can access memory and updates the base register so the next memory access is done on the updated base register. Different address modes differentiate 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. 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. The base register could be either incremented or decremented. When pre and post-indexed addressing modes are used, memory operations require at least an additional thread cycle to complete. Because the register file only contains one write port, we cannot simultaneously write back a load result from memory and the updated base register to the register file. Thus, we need to spend an extra pass through the pipeline to update the base register.

When the memory address is accessing the scratchpad memory region, memory operations can be completed in a single cycle, and the data is ready by the next (*writeback*) stage to be written back to the registers. However, if the memory read/write operation is accessing the memory region of the DRAM, the request must go through the DRAM memory controller to access the DRAM. DRAM operations typically take three or four thread cycles to complete. As discussed in chapter 2, our thread-interleaved pipeline implementation does not dynamically switch threads in and out of execution when they are stalled waiting for memory access to complete. Thus, the when a memory instruction accesses the DRAM memory region, the same instruction is replayed by withholding the update for the next PC, until the data from DRAM arrives and is ready to be written back in the next stage. For memory instructions accessing I/O regions, the access latency depends on the I/O accessed and the connection of the bus. As mentioned in section 3.2, the actual access time to I/O devices is device dependent, and a discussion of time-predictable buses is outside the scope of this thesis. In the hardware implementation, for memory instructions that access the DRAM or I/O region, it is possible to update the base register earlier during the cycles where the instruction is waiting for access to complete. However, the current PTARM implementation uses the same logic and datapath for all memory accesses (scratchpad, DRAM, I/O etc) to minimize hardware resources, so an additional cycle is used to update the base register for all memory accesses regardless of the address region they are accessing.

Load/Store Multiple

The load/store multiple instruction is used to load (store) a subset, or possibly all, of the general purpose registers from (to) memory. This instruction is often used to compact code that pushes or pops registers from the program stack. The list of registers that are used in this instruction is specified in the register list as a 16 bit field in the instruction. The 0th bit of the bit field representing R0 and the 15th bit representing R15. A base register supplies the base memory address that is loaded from or stored to, which then is sequentially incremented or decremented by 4 bytes for each register that is operated on. Figure 3.9 shows how the load/store multiple instruction is executed in the pipeline.

The load/store multiple instruction is inherently a multi-cycle instruction, because each thread cycle we can only write back one value to the register or store one value to memory. Thus, the execution state and remaining register list of the load/store multiple instruction is stored in thread state. After the decoding the instruction, the remaining thread cycles load the register field from the thread state and clears it as registers are being operated on. The instruction completes when all registers have been operated on. Each iteration the *register address decoder* in the pipeline decodes the register list and determines the register being operated on. For load multiple, this indicates the

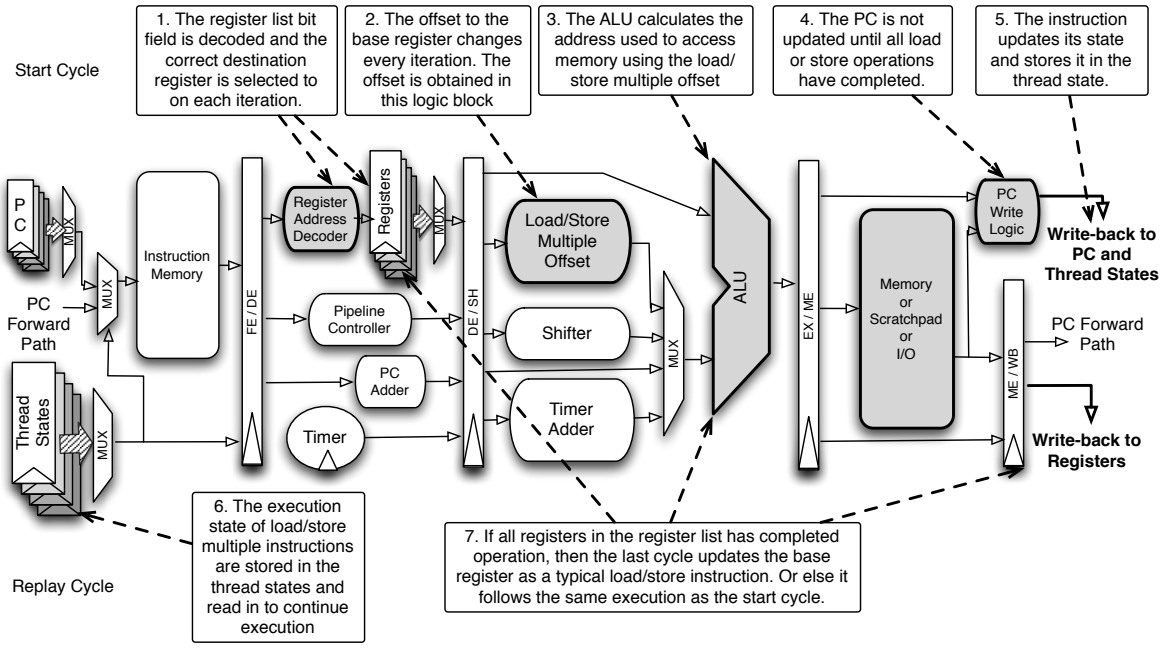


Figure 3.9: Load/Store Multiple Instruction Execution in the PTARM Pipeline

destination register that is written back to. For store multiple, this indicates the register whose value will be stored to memory. The *load/store multiple offset* block is used to obtain the current memory address offset depending on how far we are in the execution of this instruction. The offset is added to the base register to form the memory address fed into memory.

The execution time of this instruction depends on the number of registers specified in the register list and the memory region that is being accessed. For accesses to the scratchpad, each register load or store takes only a single cycle. However, if memory accesses are to the DRAM region, then register load/store will take multiple cycles. It is also possible for the load/store multiple instruction to update the base register after all the register operations completes. Similar to the load/store register instruction, an additional thread cycle will be used to update the base register. Although the execution time of this instruction seems to be dynamic depending on the number of registers specified in the register list, but the instruction binary will allow us to statically determine that number by parsing the bit field of the instruction. Thus, the execution time of this instruction can still be statically analyzed.

Load to PC

When load/store operations load to the destination register R15, it triggers a branch in the pipeline. This also holds true for the load multiple instruction if the 15th bit is set in the register list. In our five stage pipeline, we commit the next PC in the memory stage so the next instruction fetch from the same thread can fetch the updated PC. However, when the branch target address is loaded from memory, the address is not yet present in the memory stage to be committed, but only at the beginning of the writeback stage will it be present. Thus, we introduce a forwarding path that forwards the PC straight from the writeback stage to the fetch stage. Figure 3.10 shows how this is implemented in our pipeline.

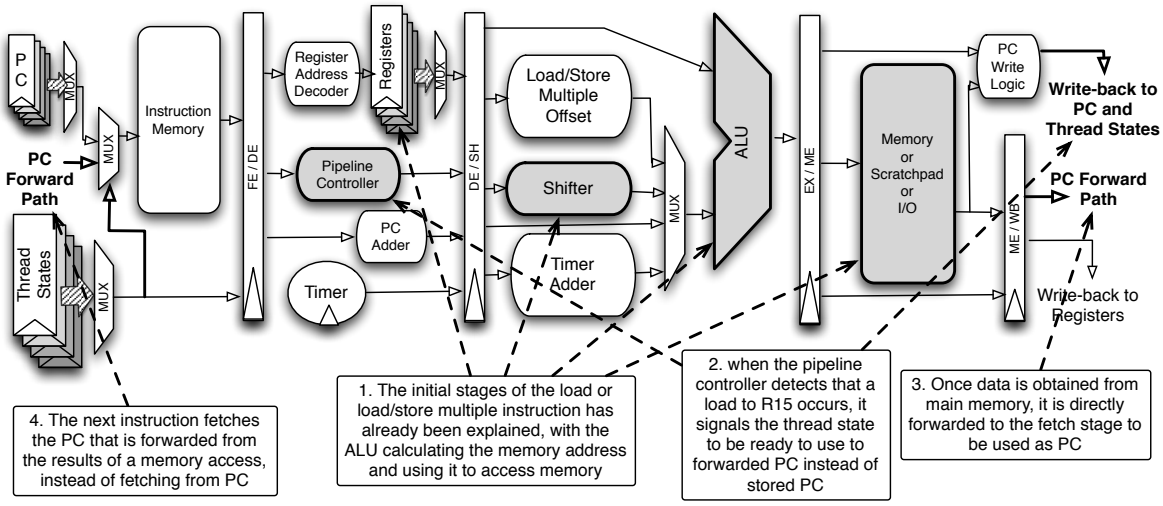


Figure 3.10: Load to R15 Instruction Execution in the PTARM Pipeline

An extra multiplexer is placed in the fetch stage before the instruction fetch to select the forward path. When a load to R15 is detected, it will signal the thread state to use the forwarded PC on the next instruction fetch, instead of the one stored in next PC. Because the data from memory will be ready at the beginning of the writeback stage, the correct branch target address will be selected and used. We discussed in section 2.1.1 the timing implications of data-forwarding logic in the pipeline. Those same principles are applied in this situation. Although it seems the selection of PC is dynamic, but when forwarding occurs is actually static; this PC forwarding only and always occurs when instructions load from memory to R15. This mechanism has no additional timing effects on any following instructions, as no stalls are needed to wait for the address to be ready. Even if the load to R15 instruction is accessing the DRAM region, the timing 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 load instruction that does not load to R15 also needs to wait until the DRAM access completes before the thread fetches the next instruction. So this extra forwarding mechanism does not cause load to R15 instructions to deviate from other load timing behaviors.

If the load to R15 instruction updates the base register, then the forwarding path is not used and not needed. The extra cycle used to update the base register will allow us to propagate the results from memory to be committed in the memory stage. The timing behavior still conforms to a regular load to registers instruction.

3.4.4 Timing Instructions

In section 2.3 we presented various instruction extensions to the ISA to bring timing semantics at the ISA level. We will now present a primitive implementation of those instructions in PTARM. ARM provides extra instruction encoding slots to be used to implement instructions for co-processors attached to the core. In our case, we implement our timing instructions as part of co-processor 13. We have already described the functionality and use case of the different timing instructions, table 3.1 shows a summary of the instructions and their op codes. All instructions

Type	Opcode	Functionality

get_time	8	offset = (crm << 32) + crn;
		deadline = *current_time* + *offset*;
		crd = low32(deadline);
		crd+1 = high64(deadline);

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

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

Table 3.1: List of assembly deadline instructions

have the assembly syntax "***cdp, p13,*** <***opcode***> ***rd, rn, rm, 0***", with <opcode> differentiating the instruction type.

The timing instructions uses a master clock to obtain and compare deadlines. PTARM implements the clock in the *timer* block that is shown in figure 3.1. Time is currently represented as an unsigned 64 bit value with nanoseconds as its units, and starts at zero when PTARM is reset. Unsigned 64 bits of nanoseconds can represent time up to approximately 584 years. (Todo: discuss timer implementation relative to different clock speeds.) Each of the timing instructions operate on 64 bit values, which is stored in two 32 bit registers. Because the deadlines are stored in general purpose registers, standard arithmetic instructions can be used to manipulate the values. However, PTARM does not current provide 64 bit arithmetic operations, so programmers must handle the overflow in software.

For each thread, the timer value is latched into the pipeline at the decode stage, which is where each thread's reference to time is. Each thread operates on their own private deadlines, and are not affected by the timing instructions from other threads. PTARM contains 4 hardware threads that are interleaved through the pipeline, so each hardware thread can only access the timer once every 4 processor clock cycles, the granularity of time observed by each thread. We will discuss the timing implications of this in section 3.5.1, in this section we merely present how they are implemented in the pipeline.

Get_Time

The *get_time* instruction is used to obtain the current timer value and store it in two general purpose registers. The *get_time* instruction also takes two optional source operands to calculate an offset to the current timer value. This allows the programmer to obtain the desired deadline time without additional arithmetic instructions. Figure 3.11 shows how *get_time* is implemented in the PTARM pipeline.

The timer value and source registers are read in at the decode stage. The *timer adder* adds the lower 32 bits of the timer value and source operand while the ALU computes the upper 32 bits taking into account the carry of the *timer adder*. Once the new deadline has been calculated, it is loaded back into the register file. Because our register file only contains one write port, so this

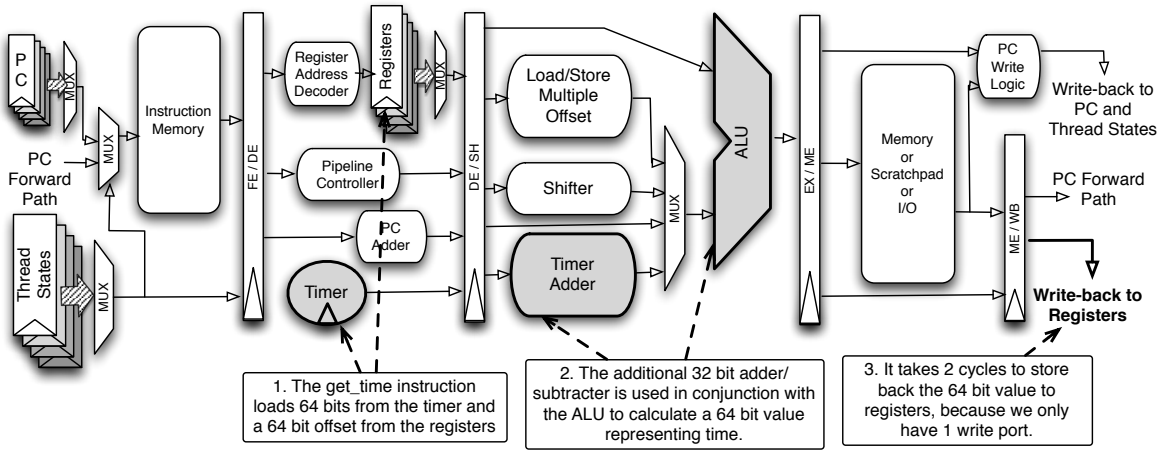


Figure 3.11: Get_Time Instruction Execution in the PTARM Pipeline

instruction also takes two thread cycles to complete; each cycle writes back 32 bits of the new value. The calculated 64 bit deadline value is written 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.

(Todo: Talk about whether or not the time passed back adjusts for the latency of the instruction.)

Delay_Until

Delay_until is used to delay the thread until the a specified deadline has been reached. It takes in 2 source operands which forms a 64 bit deadline value that is checked against the timer every thread cycle. The 2 source operands are usually the results of a *get_time* instruction. As described in section 2.3, the *delay_until* instruction can be used to specify a lower bound execution time for a code block. This could be useful for synchronization between tasks or communicating with external devices. Figure 3.12 shows the implementation of the *delay_until* instruction in the PTARM pipeline.

The implementation of the *delay_until* instruction is very straightforward, but highlights the reason the *timer adder* is added into the pipeline. The source operands and the timer value is latched at the decode stage, then compared using the *timer adder* and ALU. The next PC is only updated if the timer value is greater than the deadline values passed in. Without the additional *timer adder* in the pipeline, comparing 64 bits using our 32 bit ALU would take two thread cycles. This would decrease the precision of this instruction by a factor of two, because now we can only check the deadline against the timer every two thread cycles. The added *timer adder* allows *delay_until* to check the deadline every thread cycle, to ensure that no additional threads cycles have elapsed right after the deadline is reached.

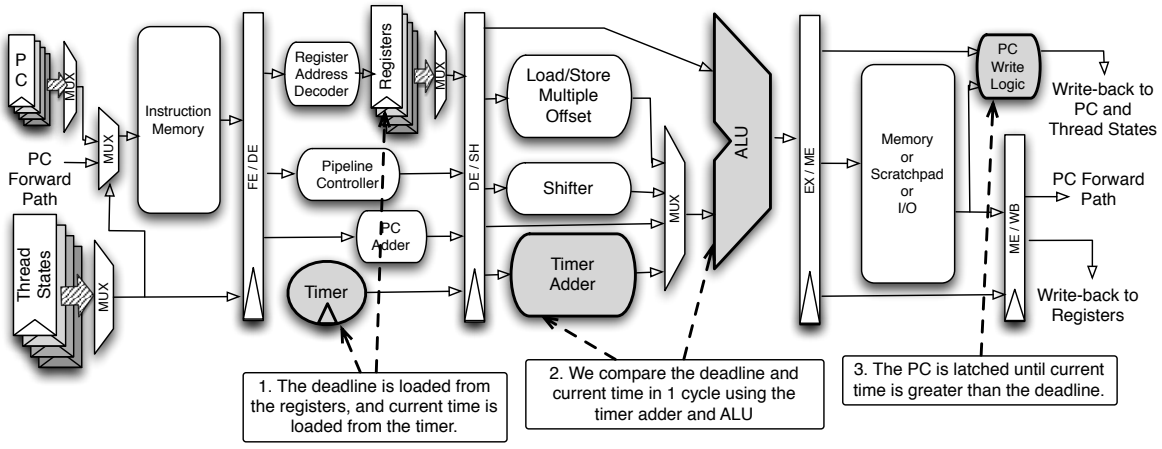


Figure 3.12: Delay_Until Instruction Execution in the PTARM Pipeline

Exception_on_Expire and Deactivate_Exception

Exception_on_expire and *deactivate_exception* provide a mechanism to actively check for missed code that runs longer than a specified deadline. *Exception_on_expire* is used to register a deadline for immediate miss detection. *Deactivate_exception* is used to deactivate the active checking before the deadline expires. Unlike the *delay_until* instruction, which checks the deadline when the instruction is decoded, the deadlines registered with *exception_on_expire* are checked in hardware in the background. A timer expired exception is triggered in the pipeline when a missed deadline is detected. In section 2.3 we have outlines examples of how the instructions are used.

The actual execution of the *exception_on_expire* and *deactivate_exception* instructions is straightforward. Within the *timer* unit, there is one 64 bit deadline slot for each thread to register an actively checked deadline. With four threads in PTARM, there are four slots in the *timer* hardware. Whenever an *exception_on_expire* instruction is executed, two source operands are read in and stored to the thread's corresponding deadline slot in the *timer*. Every clock cycle of the timer, active deadlines are checked against the current timer value. Once the current timer value surpasses one of the active deadlines, a timer expired exception for the particular thread is raised in the pipeline. We add an entry to the existing ARM exception vector table to create this timer expired exception. The exception is handled the same as any other exception, as discussed in section 3.3, and is only handled by the thread that registered the deadline. The other threads in the pipeline remain temporally isolated from this exception. The execution of a *deactivate_exception* instruction simply clears the deadline slot for the specific thread.

If threads need to simultaneously check for multiple deadlines, then the single deadline slot for the thread needs to be managed in software. The software overhead involves managing a list of deadlines and ensuring that the earliest deadline is always being checked in the timer. It is possible to implement more than one deadline slot for each thread in the timer if more precise deadline checking is needed. However, this comes at the cost of additional hardware complexity in the timer, so currently in PTARM we simply have one deadline slot per thread, and use software mechanisms to manage multiple deadlines in a thread.

3.5 Timing Analysis of PTARM

3.5.1 Precision of timing instructions

3.6 PTARM VHDL Soft Core

Our pipeline can be clocked up to $100MHz$ when synthesized to a Virtex-5 lx110t FPGA.

Figure 3.13 shows the high level block diagram of the Softcore.

Talk about I/O devices, including UART, DVI controller and Interface with DDR2 DRAM controller

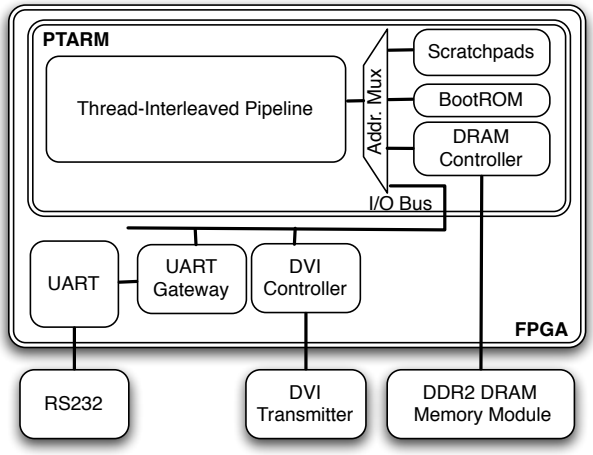


Figure 3.13: PTARM Block Level View

3.7 PTARM Simulator

Talk about experimentation with DMA and memory hierarchy

Chapter 4

Applications

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

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

4.1 Real-Time 1D Computational Fluid Dynamics Simulator

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

or damped before subsequent injection events to ensure the correct amount of fuel is injected [15]. Currently, fuel rails are modeled and developed with 1D-CFD solvers like GT-Fuel, and use an ad-hoc model of fuel pressure for injection events [99]. 1D-CFD models are commonly used in simulating transient operation of internal combustion engines [82]. Here, we present an implementation for real-time execution of a 1D-CFD solver using multiple PRET cores that model the fuel rail. Although the calculations are slightly rougher than the GT-Fuel calculations, it is sufficient to allow improved fuel pressure estimation and close the loop of fuel delivery, allowing for a cleaner, more efficient engine.

4.1.1 Background

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

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

In this equation, a is the wave speed, C is the Courant number and Δx is the descretization length. For stability, the Courant number needs to be less than 1 and a number below 0.8 is recommended [31]. 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 and a 1 cm discretization length is common for diesel fuel systems. For diesel engines, a speed of sound (wave speed) of 1500 m/s [86] is commonly used. 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 [104], such as in image processing applications.

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

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

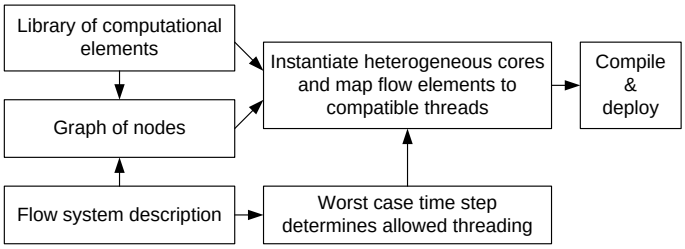


Figure 4.1: Design Flow

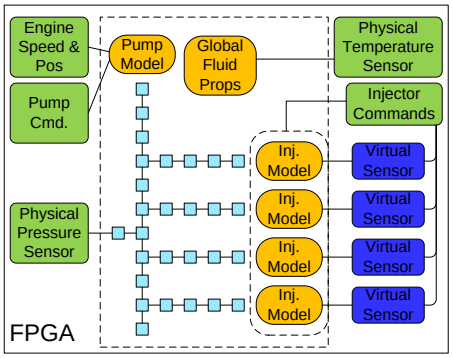


Figure 4.2: High Level System Diagram

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 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 [91].

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

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

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

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

For illustrative purposes, we show a sample pipe network graph in figure 4.3. Each pipe element is also referred to as a computational node. Its graphical representation is shown in Table 4.4. This pipe network starts with an imposed flow input (P1) element on the left, which represents a pump. Fluid travels through a few pipe segment nodes (P2 and P3) to a "T" intersection (P4), where it splits off to a second branch of the network. The "T" node is also measured by the outside world

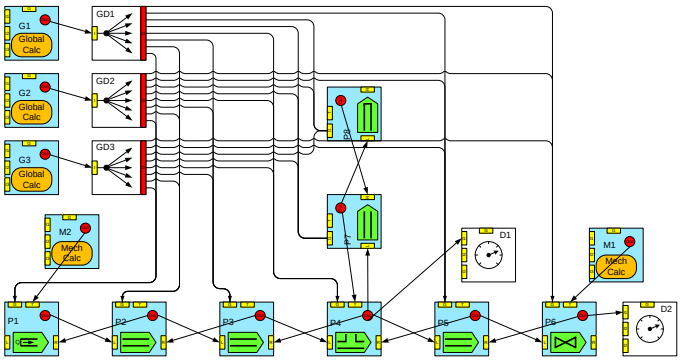


Figure 4.3: Detailed System Diagram

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

(D1) through a output port. Output elements are used when data needs to be communicated out of the model to other parts of the FPGA. Flow going up the new leg ends in a cap (P8), while flow continuing down the original path exits the system through a valve (P6). *Mechanical calculation* elements compute the inputs to valve, defined flow, and defined pressure blocks. The system is assumed to be at uniform temperature. Temperature dependent variables like density and wave speed are computed by the global calculation nodes (G1, G2, and G3). This values are needed by all computational elements in the graph, thus are distributed by the global distributions (GD1, GD2, and GD3) to each of the computational elements every time step.

4.1.2 Implementation

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

Hardware Architecture

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

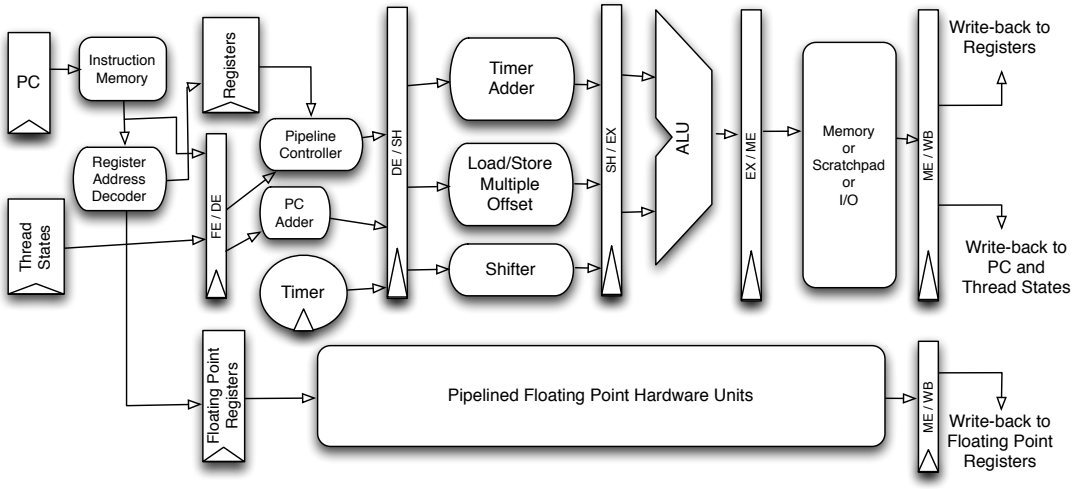


Figure 4.5: The PTARM 6 Stage Pipeline

pipeline also contains hardware floating point units to support the applications needs of floating point computations. The floating point units are single-precision, and generated using the Xilinx Coregen tool (Todo: cite). They are pipelined to accept input every cycle, which avoids structural hazards, as explained in section 2.1.3. The floating point operations supported are: add, subtract, multiply, float-to-fix, fix-to-float, divide and square root.

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

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

subtractions appear as single thread cycle instructions because their latencies are fully hidden by the thread interleaving.

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

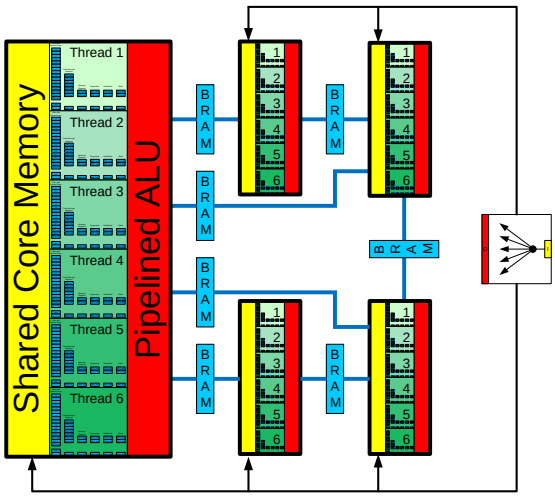


Figure 4.6: System of PRET Cores and Interconnects

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

When implementing the global distribution circuit, we observed that only a few nodes are required to the broadcast all the temperature dependent parameters. In fact, in diesel fuel systems, the number of nodes needed to broadcast all parameters can be mapped to the six threads of one single PTARM core. Thus, we dedicate one PTARM core in the system as the broadcast core. For each other core, we add a dedicated broadcast receiving memory that is connected to the broadcast core. The broadcast receiving memory is also synthesized to a small dual-port BRAM, with a read-only port connected to the core, and a write-only port connected to the broadcaster. The broadcast core contains a broadcast bus that can simultaneously write to all the broadcast memories the same

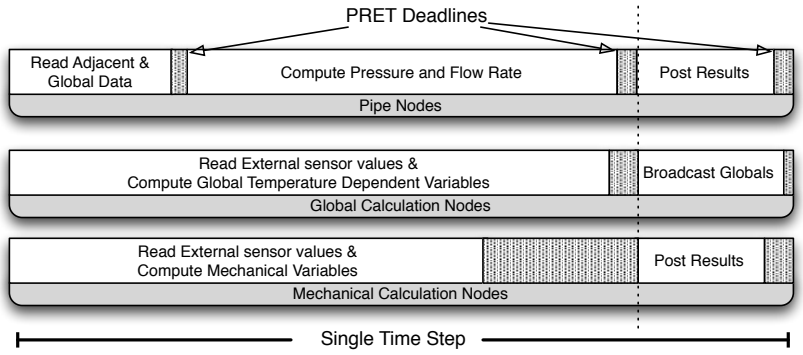


Figure 4.7: Execution of Nodes at Each Time Step

values. The broadcast memory is also shared amongst all threads in a core so all threads can access the global values. This architecture allows us to save on the resources needed to implement a full fledged interconnect routing system or any network protocol to be used for broadcasting. Figure 4.6 shows a block-level view of the hardware architecture.

Software Architecture

We implement the equations in table 4.1 in the language C and compile it with the GNU ARM cross compiler [1] to run on our cores. In order to minimize the computation required, the equations are statically optimized. The communication channels in and out of each node is memory mapped to the shared BRAMs between cores.

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

In section 2.3 we introduced ISA extensions that provide programmers with explicit timing control in software. The implementation of the various timing instructions for PTARM is explained in section 3.4. Specifically for this application, we use the specialized timing instruction *delay_and_set* as introduced in section 2.3. The semantics of the *delay_and_set* instruction is similar to the deadline instruction introduced by Ip and Edwards(Todo: cite). When it is decoded, it first enforces the previously specified timing constraint, then it sets a new timing constraint for the next

code block. The instruction enforces a minimum execution time within the code, which we use to enforce the synchronized execution of time steps for all nodes. Fig. 4.7 shows the program synchronization points that our timing instruction enforces. The hatched area in the figure denotes slack time that is generated by the timing instructions. Each *delay_and_set* instruction takes 2 thread cycles because it manipulates a 64-bit value representing time. For our computational nodes, 3 timing instructions are used each time step, thus 6 thread cycles of overhead are introduced per time step.

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

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

4.1.3 Experimental Results and Discussion

We use three examples to evaluate our framework. The first example is a simple water-hammer example taken from Wylie and Streeter [100]. It is similar to the one shown in figure 4.3, but without the “T” element and the nodes that branch up. This example contains an imposed pressure, 5 pipe segments, a valve, and two mechanical input blocks that provide both the reference pressure and the valve angle as a function of time. We use this simply as a sanity check for the

correctness of functionality of our framework.

The second and third example cover two common diesel injector configurations: the unit pump and common rail. The data for configuring these cases was taken from reference examples provided by Gamma Technologies' GT-SUITE software package [31]. 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 complex where the topology is roughly that described by the 1D-CFD model in figure 4.3. It has a total of 234 sub-volumes, including 5 "T" intersections and 4 valves. Both the GT-SUITE-based models use a 1 cm discretization length, which, using a 1500 m/s wave speed, and a stability factor of 0.8 yields a 5.33μs 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 [102] with speed grade 3. Each Virtex-6 FPGA logic slice contains 4 LUTs and 8 flip-flops, and this FPGA contains 31,200 logic slices and 512 18-KB BRAMs. Each PRET core is clocked at 150 MHz and has 6 threads. All floating point units are generated from the Xilinx Coregen tool [101], and are configured to maximize DSP slice usage and minimize logic slice usage as much as possible. We save the logic slices to synthesize as many cores as possible. Our current PRET implementation, the PTARM, uses an ARM-based ISA, thus our C code is compiled using the GNU ARM cross compiler [1] with the optimization compiler flag set to level 3. For these examples, we used a mapping heuristic that grouped nodes requiring same computations onto the same core. In the sections below we will show that this heuristic allows us to save hardware resources by synthesizing less floating point units.

Timing Requirements Validation

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

To convert thread cycles to physical time, we use the processor clock speed and number of threads executing in the architecture. Given a 150 MHz clock rate and six hardware threads, each thread executes at 25 Mhz in our thread-interleaved pipeline. Thus, each thread cycle converted to physical time is 40 ns long. The unit pump and common rail have a requirement of 5.33 μs, which gives us 133 thread cycles to complete the computation each time step. Table 4.2 shows that

	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

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 that of the two complex examples. This validates that we can safely meet the timing requirements, ensuring the correctness of functionality of our implementation.

Resource Utilization

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

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

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

Two important observations are made from the results of table 4.3. First, the area increase associated with adding more threads to the core is proportional only to the number of bits required to encode the number of threads. For example, running 6 threads or 8 threads (both requiring three bits to encode the thread number) on the processor yields a similar area usage. But once a 9th thread is introduced, the used area noticeably increases, but remains similar for up to 16 threads. This can be explained by understanding the architecture of multi-threaded processors. Multi-threaded processors maintain independent register sets and processor states for each thread, while sharing the datapath and ALU units amongst all threads. The register sets are synthesized onto BRAMs, so

the number of bits used to encode thread IDs will determine how big of a BRAM is used for the register set. The size of the multiplexers used to select thread states and registers is also determined by the number of bits encoding the thread IDs, not the actual number of threads running. Thus, it is possible to increase the number of threads per core with almost negligible impact on area as long as the incremented thread count uses the same number of bits to encode. Increasing the thread capacities will allow our architecture to support more nodes in a single FPGA. However, since hardware threads share the processor pipeline, adding threads slows down the running speed of the individual threads. Nonetheless, for implementation that have sufficient slack time or require faster performance, adjusting the number of threads could lead to a valuable improvement. Our precise execution time analysis allows us to determine the maximum number of threads, six in our case, we can support to meet our timing constraints. An over estimated execution time in this case could lead to under utilizing the hardware by constraining the number of threads to five, resulting in requiring additional cores to implement our 237 node fuel rail example.

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

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

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

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

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

far below the logic cell utilization, validating our design choice of using BRAMs for interconnects and broadcasts.

4.1.4 Conclusion

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

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

4.2 Eliminating Timing Side-Channel-Attacks

Encryption algorithms are based on strong mathematical properties to prevent attackers from deciphering the encrypted content. However, their implementations in software naturally introduce varying run times because of data-dependent control flow paths. Timing attacks [44] 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 [44]. 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 [21, 26, 105] that clearly indicate the "noise" by itself is insufficient protection. In fact, the architectural unpredictability that was initially believed to prevent timing attacks was discovered to enable even more attacks. Computer architects use caches, branch predictors and complex pipelines to improve the average-case performance while keeping these optimizations invisible to the programmer. These enhancements, however, result in unpredictable and uncontrollable timing behaviors, which were all shown to be vulnerabilities that led to side-channel attacks [16, 67, 3, 24].

In order to not be confused with Kocher's [44] 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 [54, 43], fault injection

attacks [18, 29], and many others [105].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [18, 24, 45, 40, 2, 41, 23, 93, 92]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-channel exploits [24, 18, 45, 40, 2, 41, 23, 93, 92] indicates that we do not have the complete set of solutions as more and more vulnerabilities are still being discovered and exploited. Just recently, Coppens et al. [24] 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 [44] or blinding signatures [44, 23]. Other techniques such as branch equalization [58, 105] use software techniques to rewrite algorithms such that they take equal time to execute during each conditional branch. We take a different approach, and directly address the crux of the problem, which is the *lack of control* over timing behaviors in software. We propose the use of an embedded computer architecture that is designed to allow predictable and controllable timing behaviors.

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

For this application, we present PRET in the context of embedded cryptosystems, and show that an architecture designed for predictability and controllability effectively eliminates all time-exploiting attacks. We target embedded applications such as smartcard readers [45], key-card gates [20], set-top boxes [45], and thumbpods [77], which are a good fit for PRET's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [61] encryption algorithms on PRET, and show its immunity against time-exploiting attacks. This work shows that a disciplined defense against time-exploiting attacks requires a combination of software and hardware techniques that ensure controllability and predictability.

4.2.1 Background

Kocher outlined a notion of timing attacks [44] on encryption algorithms such as RSA and DSS that require a large number of plaintext-ciphertext pairs and a detailed knowledge of the target implementation. By simulating the target system with predicted keys, and measuring the

run time to perform the private key operations, the actual key could be derived one bit at a time. Kocher also introduced power attacks [54, 43], 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 [60, 41], which also found side-channel attacks against IDEA, RC5 and blowfish [41]. Fault-based attacks [18, 40, 29] were introduced by Bihan et al. [18]. These attacks attempt to extract keys by observing the system behavior to generated faults. For the side-channel attacks that we have missed, Zhou [105] presents a survey on a wide range of side-channel attacks.

Dhem et al. [26] demonstrated a practical implementation of timing attacks on RSA for smart cards and the ability to obtain a 512-bit key in a reasonable amount of time. Several software solutions such as RSA blinding [44, 23], execution time padding [44], and adding random delays [44] have been proposed as possible defenses against this attack. However, these solutions were not widely adopted by the general public until Brumley et al. [21] 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 [29] and AES [16]. In particular, Bernstien's attack on AES [16] 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 [67] 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 [3, 2] that monitor control flow by occupying a shared branched predictor. Compiler and source-to-source transformation techniques [24, 58] have also been developed to thwart side-channel attacks.

Wang et al. [92] 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 [93] and Random Permutation caches [92] that defend against caching attacks in hardware. Very recently, Coppens [24] discovered two previously unknown attacks on the complex pipeline run time variance of x86 architectures.

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

4.2.2 A Precision Timed Architecture for Embedded Security

The foundation of time-exploiting attacks exploits the uncontrollable timing variability introduced to programs by underlying the implementation of encryption algorithms. Software implementations naturally introduce varying run times because of data-dependent control flow paths. Modern computer architectures create unpredictable execution times by abstracting away hardware optimizations meant to improve average case performance. In this section we will present several features of PRET that bring *controllability* over timing to software, eliminating the origin of the attacks. We will discuss the software extensions that allow timing specification in programs, and the predictable architecture to comply with these specifications. These two approaches cannot be

separated. A predictable architecture by itself would only ease the feasibility of an attack, and software timing specifications are meaningless if they cannot be met by the hardware. By combining both hardware and software solutions, we yield a timing predictable and controllable architecture. Thus, by design, PRET prevents leakage of any timing side-channel information, and eliminates the core vulnerability of time-exploiting attacks.

Controlling Execution Time in Software

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

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

At the expense of more programming effort, other solutions have been proposed to alter and pad the execution time of different execution paths [44] to shield against the timing variability of the algorithm. At a glance it might seem that the timing instruction are a similar solution to these proposals, however, the principles are inherently different. While effective against certain time-exploiting attacks, existing solutions alter the underlying algorithm implementation in attempt to manually pad or distort the execution time. These solutions are not only algorithmically specific, but could lead to unnecessarily degrading of the performance of encryption algorithms. The timing instructions, on the other hand, allows for a separation of concern between the functionality and timing behavior of the code. The programmer can implement the correct functionality of the algorithm, then use timing instructions to regulate its timing behavior. The subtle difference will be more apparent in section 4.2.3 when we show two different implementations of the RSA encryption that both use timing instructions to regulate execution time. One implementation mimics

existing execution time padding solutions, and the second implementation uses timing instructions to enforce an overall execution time of the RSA algorithm. We present performance comparisons and show that explicit timing control instructions could prove more beneficial than simple execution time padding.

The timing instructions provide a method to control the timing behavior of a program in software. However, they do not change the behavior of the underlying architecture. If the underlying architecture makes the reasoning of execution time difficult, then these instructions become more difficult to use. Timing instructions alone do not prevent attacks that exploit architectural designs to inject execution time variances [67, 2] and obtain side-channel information. We argue that a *predictable* architecture is also required to eliminate timing exploiting attacks.

Predictable Architecture

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

Computer architects use clever speculative techniques to mitigate the effects of pipeline hazards and to substantially improve the average-case performance. For example, branch predictors are used to guess the next instruction needed by the processor for branches [32]. This allows the processor to execute instructions speculatively while rolling back only when needed. While these speculative techniques improve the average-case performance, they introduce several side effects. First, they create *timing variations*. Depending on the outcome of its speculation, the processor might need to discard the wrongly speculated work, and re-execute the correct instructions. Second, these units are *unpredictable*. Since these units are shared by all software processes concurrently running on the processor, the states of speculation units are heavily dependent on the different interleaving of processes. This means that a process can unknowingly be affected by other processes, since the speculation state is shared between them [50]. Because the goal of these speculation techniques is to improve program performance without effort from the programmer, the controls of these speculation units are concealed from the programmer, and cannot be directly accessed in software. Thus, these side effects result in *uncontrollable* timing behaviors in the program.

Several Multithreaded architectures enable more opportunities to exploit the uncontrollable timing behaviors. Multithreading utilizes thread-level parallelism by introducing multiple hardware threads in the processor. This allows the execution of another hardware thread during pipeline stalls like branches or memory accesses. However, typical multithreaded architectures share hardware units effecting execution time between hardware threads, enabling threads to covertly affect other threads execution time. The class of Simultaneous Multithreading (SMT) architectures presents an example of this. Here, the hardware threads share multiple execution units and execute in parallel depending on a hardware scheduler. Attackers exploit such designs by running a spy thread that executes concurrently with a thread that implements the encryption algorithm. This spy thread probes the components shared with the encryption thread [67, 2] by forcefully occupying the shared units and observing when they are evicted by the encryption thread. The announcement of this vulnerability caused Hyper-Threading, Intel's implementation of SMT, to be disabled by

default in some Linux distributions because of its security risks [68]. For general purpose applications, these side effects pose insignificant threats, but for security applications, the consequences are uncontrollable sources of side-channel information leakages.

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

Memory System The memory system presents another opportunity for attackers to gain side-channel information. The high clock speed of modern processors combined with the high latency to access main memory results in sometimes hundreds of cycles stalled when the processor needs to access the main memory. On-chip fast access memories are used to bridge this access latency, creating a *memory hierarchy*. Caches are *hardware-controlled* fast-access memories that predict and prefetch data from main memory based on temporal and spatial locality of data accesses from the processor. If the cache control speculation is accurate, then access to data can complete in one cycle, and no stall in the pipeline is required. However, when a misprediction occurs, data needs to be fetched from the main memory, causing a drastic difference in the access time [87]. Caches abstract away this memory hierarchy and access latency variation from the programmer by managing the cache contents in the hardware. Because threads and processes share the same memory system, attackers can probe the memory access patterns of the encryption process by evicting shared cache lines and observing the timing variation it causes [67]. This is possible because the memory hierarchy is abstracted away from the programmer, resulting in *uncontrollable* timing behaviors.

PRET utilizes scratchpads memories (SPM) instead of caches in its memory hierarchy. SPMs are fast access memories controlled by software. SPMs use less power and occupy less area [11] because no speculation logic is needed. SPMs occupy a distinct address space, which exposes the memory hierarchy to the programmer, instead of abstracting it away like caches. The allocation of data between memory and SPM is done with explicit instructions, either at compile time by the compiler or manually by the programmer. This gives the software control over memory access latencies, and provides a predictable execution time of the program. The performance of SPMs vary based on the data access patterns of the application, but since the control is in software, it is possible to tune the allocation scheme to achieve even better performance than a generic cache for specific applications. There are abundant ongoing research on allocation schemes and meth-

ods for optimizing the performance of SPMs [9, 13, 64, 85, 88]. SPMs can be found in the Cell processor [33], which is used in Sony PlayStation 3 consoles, and NVIDIA's 8800 GPU, which provide 16KB of SPM per thread-bundle [62]. Although this comes at the cost of more programing effort, but it is not uncommon to see platform specific tuning of software for performance purposes. For example, high performance parallel algorithms are often fine tuned to work on block sizes depending on the cache size and replacement policy of the platform, and re-tuned when running on different platform.

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

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

We acknowledge the many efforts to counteract timing attacks with algorithm rewrites to control and balance the run time of the algorithm. These efforts while successful, are ad-hoc, counteracting specific attacks without prevention of others. Without tackling the origin of time-exploiting attacks, we believe that more exploits will eventually be discovered, attacking the *uncontrollable* execution time variation caused by the shared resources of hardware or software control flow. The PRET architecture is designed to ensure repeatable and predictable timing behavior of programs by providing control of timing properties in software and a predictable architecture that provides temporal isolation for hardware threads and processes. PRET is impenetrable known attacks such as branch predictor attacks [3], cache attacks [67] or other attacks on the pipeline [24]. The more importantly, the predictable architecture design removes the root cause of time-exploiting attacks – the *uncontrollable* timing variations caused by unpredictable hardware components or software control flows.

4.2.3 Case Studies

In the following section we will show results of two encryption algorithms running on PRET. All experiments are run on the cycle accurate simulator of the PRET architecture described in [52]. The simulator implements the SPARC v8 instruction set, and employs six threads on a six stage thread-interleaved pipeline. Programs are written in C and compiled using a standard gcc cross compiler from Gaisler research labs [30]. This PRET implementation implements a simple processor extension inspired by Ip and Edwards [37] that adds timing instructions to the ISA. To be consistent with the terminology used in [37], we call this instruction the *deadline instruction*. This deadline instruction has similar semantics to the *delay_and_set* instruction introduced in section 2.3. It first ensures the previous deadline specified is met, then sets the deadline for the next instruction sequence. The deadline instruction specifies time in units of thread cycle, which is a thread's perceived cycle.

RSA Vulnerability

The central computation of the RSA algorithm is based primarily on modular exponentiation. This is shown in algorithm 1. Of the inputs, M is the message, N is a publicly known modulus, and d is the secret key. Depending on the value of each bit of d on line 4, the operation on line 5 is either executed or not. This creates variation in the algorithm's execution time that is dependent on the key, as mentioned in [44].

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

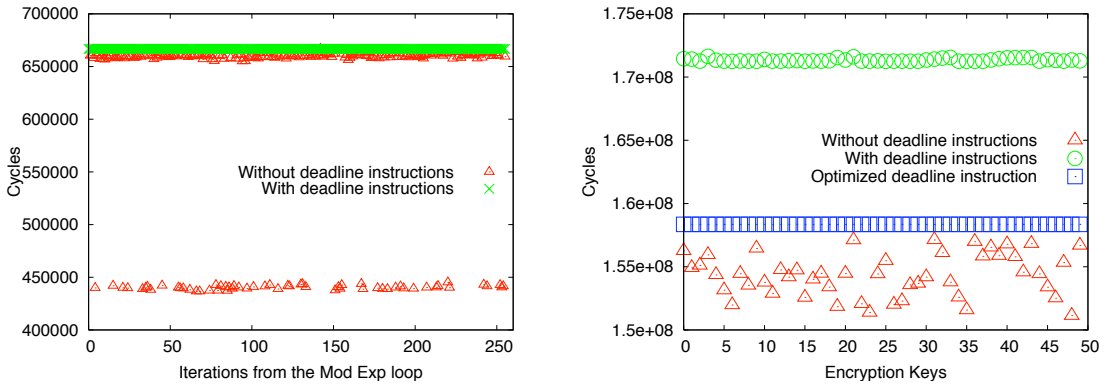
Algorithm 1: RSA Cipher

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

Algorithm 2: RSA Cipher with deadline instructions

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

We observe the large-scale effect of this small change on the whole encryption in figure 4.8(b), where RSA was run fifty times using randomly generated keys. Without the deadline instructions (triangle points), different keys exhibit significant diversity in algorithm execution time. With the deadline instructions added within the modular exponentiation loop (circle points), the fluc-



(a) Run time of Modular Exponent operation

(b) Run time of RSA operation

Figure 4.8: RSA Algorithm

tuation 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've showed. As a result, although this makes the encryption algorithm completely secure against time-exploiting attacks, they are not adopted in practice because of this overhead. Nevertheless, with control over execution time, we will show that running encryption algorithms in constant time does not necessarily require it to run at the absolute worst-case execution time.

An Improved Technique of using Deadline Instructions

It is expected that the distribution of RSA run times will be normal over the set of all possible keys [44]. 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.

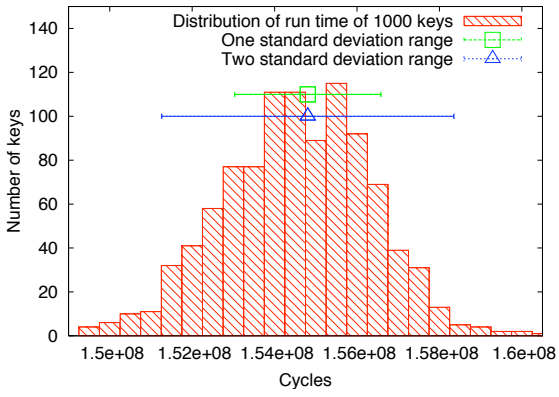


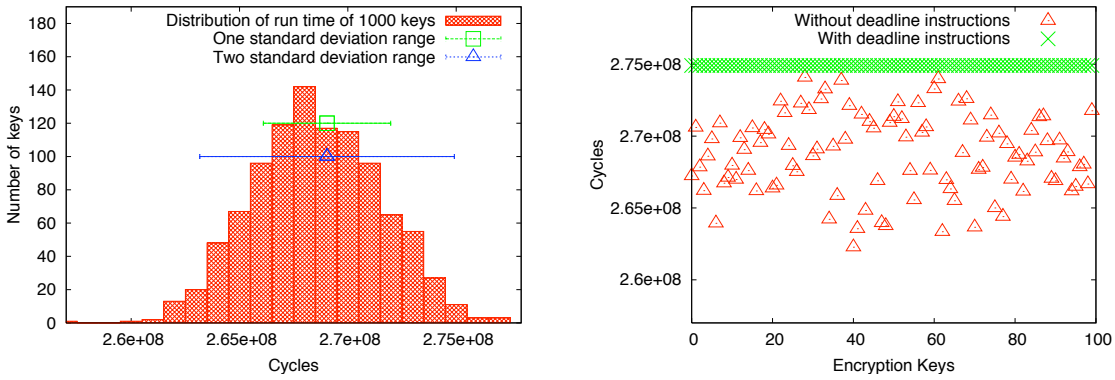
Figure 4.9: Run time distribution of 1000 randomly generated keys for RSA

First, instead of enclosing the loop itera-

tions 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 [72]. 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 anyway, since it is easier for time-exploiting attacks to distinguish those keys. By doing so, we reduce the execution time of the encryption algorithm because we know that keys that are right-side outliers will not be used.

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



(a) Distribution of 1000 DSA keys

(b) Run time of 100 DSA operations

Figure 4.10: Digital Signature Standard Algorithm

Digital Signature Algorithm

Kocher's [44] original paper mentioned that Digital Signature Standard [61] is also susceptible to timing attacks. Thus, to further illustrate our case, we ported the Digital Signature Algorithm from the current OpenSSL library (0.9.8j) onto PRET. We used the same method mentioned above to secure this implementation on PRET. Figure 4.10(a) shows the distribution of DSA run time for one thousand keys. It also shows a normal distribution. Then, we randomly generated another one hundred keys, and measured the run time with and without deadline instructions, which we show in figure 4.10(b). We can see clearly that the run time with deadline instructions is constant, and any time-exploiting attack is not possible.

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

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

4.2.4 Conclusion and Future Work

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

We demonstrate the application of these principles to known-vulnerable implementations of RSA and DSA, and show that PRET successfully defends against time-exploiting attacks with

low overhead. Our work does not undermine the significance of any related work, which have mostly been specific to certain attacks. PRET does not target a specific encryption algorithm, because it can be used in combination with these partial solutions on specific encryption algorithms, as well as provide a complete defense for other encryption algorithms which are less researched upon.

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

Chapter 5

Related Work

Craven et. al [25] implements PRET thread interleaved pipeline as an open source core using OpenFire

5.1 Real Time Adaptions

5.1.1 Branch Prediction for Real Time Purposes

Dynamic Branch predictors are a pain to model because the effect of *aliasing* of branch points. *Aliasing* occurs when two different branches occupy the same branch predictor slot, and cause interference, which most of the time is destructive, and very complex to model. They also cause timing anomalies [28]. Burguiere et al. [22] (Pascal Sainrat) made a case for *static branch prediction* to be used for real time systems. This could be done in several ways. Classic static branch predictions have relatively simple prediction schemes. One scheme is to predict either all branches are taken or not taken. Improvements include the *Backward Taken, Forward Not taken* scheme, to improve performance for loops and if statements. In Burguiere's work, they analyze different code patterns (loops, if-then-else, if then) and assign a static prediction for those patterns. This removes *aliasing* and gives better estimated worst case branch mispredicts. Architecture support is common now for static branch predictions, where compilers can insert instruction set support constructs to denote the static prediction of the branch. The underlying architecture will use that for its prediction, instead of relying on a dynamic hardware unit.

Bodin et al. [19] further pushes the idea of static branch prediction to improve WCET of programs. The idea is the use static branch prediction for the worst case execution path, and remove all branch mis-prediction penalty on the worst case path to improve the performance of the worst case path. They propose an algorithm that iterates through the CFG to find the WCEP. Initially, all the branches are assumed to be mispredicted, and the algorithm uses IPET to find the worst case path when each branch is mispredicted. Then, the algorithm assigns a static branch prediction to the branches on the WCEP by predicting the WCEP to be taken. The algorithm then iterates again to find the WCEP with the newly predicted branches. If two iterations yield the same WCEP, then the algorithm is done. Since the algorithm never reassigns assigned branches, it always converges. It's not optimal since it never reassigns branch predictions. This can effectively lower the WCET of a program, and remove the unpredictability of branches.

5.1.2 Real Time Superscalar

For superscalar processors, attempting to model all advanced techniques leads to either very pessimistic results, or almost infeasible complex models. This is because the amount of inherent state that's kept by the pipeline is large. Rochange et al. [75] (Pascal Sainrat) first proposed to fit a superscalar processor for real time purposes, by trying to ease the analysis using pre-scheduling of instructions. The concept is similar to resetting the pipeline state before each basic block execution. This is done by postponing the scheduling of the next basic block until anything from the previous basic block that can effect the timing is committed. The paper assumes no cache or TLB, nor does it mention any branch prediction, which could effect execution across basic blocks. Although there is no formal proofs in the paper, but the idea is intriguing. If one could effectively remove all timing interference across basic blocks, then the resources needed to model the pipeline could be significantly reduced, since you only need to model for a basic block, and the initial state is consistent for each basic block. However, depending on how many instructions can be in flight at one time, waiting for the pipeline state to be flushed could induce large penalties for programs with a lot of control flow transfer and small basic blocks.

A time-predictable execution mode for superscalar pipelines with instruction prescheduling [75]

- What is the background of this work? What is the motivation?

- What is the main goal?
  They want to reconcile high performance with time predictability. Mainly, they are making out of order superscalar pipelines fit WCET estimation techniques.

- What did they do to achieve this goal?
  They control instruction flow to remove dependence between basic blocks so that any WCET estimation tool would only need to measure or estimate a smaller segment of code.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They showed a performance comparison of slow down compared to regular out of order superscalar and also against in order scalar pipeline to show performance improvement.

- What other work is listed as future work?

Additional questions:

- What are the limitations/assumptions of this work?
  They ignore peripheral components (cache memories, TLBs) as well as external events (interrupts) and interactions with the operating system (process scheduling, virtual memory, etc).

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)

Predictable Out-of-order Execution Using Virtual Traces [96]

- What is the background of this work? What is the motivation?
  The motivation is to improve WCET of complex processors, specifically for out-of-order superscalar pipelines.
- What is the main goal?
  The main goal of this paper is 3 fold. 1) minimizing the pessimism introduced in WCET analysis. 2) increasing CPU throughput that can be guaranteed. 3) minimize CPU modeling cost.
- What did they do to achieve this goal?
  They introduced a VTC (virtual trace controller) to control the progress of the pipeline. They argue that this controller can be used for a CPU of arbitrary complexity. The VTC operates CPU programs as a collection of traces. Traces are paths through the program. Essentially traces are formed by statically predicting branches, in the context of this paper they predict the branches towards the worst case execution path. This way the pipeline optimizations (out of order execution etc) can optimize the worst case path. This achieves goal 2, which is increased the guaranteed throughput. The traces are formed via static branch predictions, and 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 side exits are determined by branch mispredicts. Now WCET within each trace and side exit can be measured, and it will be the same execution time, thus no CPU modeling cost is needed. This achieves goals 1 and 3.
- How do they evaluate their approach? Does it achieve the goal? They use the Malardalen WCET benchmark suite, but assume that benchmark programs are single-path programs. They assume using IPET or methods can find the WCET. They use the benchmark programs and run the program on an idealized in order machine to compare the results with their machine. They compare the speed up /slow down and use it to analyze the issues with their approach.
- What other work is listed as future work?

Additional questions:

- What are the limitations/assumptions of this work?
  They assume WCEP is easily obtained. It also depends on how many traces are formed, and how effective the traces are formed. They assume scratchpads in this work.
- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  They need a compiler to compile code into traces and form traces. The hardware is modified with a VTC to control the traces and stall the pipeline between traces.

Extending the superscalar work, Whitham et al. [96] proposed using a slightly modified out of order superscalar pipeline with *virtual traces* to provide a predictable and repeatable processor for single thread execution. [95] explains in more detail how virtual traces are formed. Basically

virtual traces utilize the idea described in Bodin et al. [19](see 5.1.1), and use static branch prediction to guide the processor along the path of a *trace*. A trace is formed using a similar algorithm in [19], except that its size (L) is determined by the number to branches it speculates. This could be limited by the complexity of the algorithm, although there is usually an optimal point in which you won't get more parallelism even if you pass it. In the out of order pipeline, fetching of the instructions of a trace is stalled until the previous trace is completely out of the pipeline. This allows traces to be analyzed starting from a fresh state in the pipeline, and prevent interference between traces. There is also constrains on other unpredictable events that might be derived from some pipeline state which cannot be flushed (padding variable length instructions (what do these instructions depend on?), disallowing memory prediction assuming scratchpads and disallowing branch instructions to be reordered (so you will only be in one trace, as branches represent exits to the trace) etc). This allows for the execution of traces to run at "constant" time for each different exit (the worst case time is the main path of the trace). By doing this you remove the need for WCET analysis because you can just do it by measurement.

There are issues with this work, as the assumption is the ability to find the WCEP when doing the trace scheduling (how do you obtain numbers for the basic blocks with Out of Order execution, and what if program itself is so complex that the analysis is nearly infeasible?). But the idea of using static branch prediction in combination with WCET could be leveraged. The delayed scheduling of traces to flush the pipeline could be a huge penalty for programs with small tasks that execute frequently. Reducing this delay is thus a trade off between performance of traces vs amount of state to keep to obtain the performance.

5.1.3 Real Time VLIW

VLIW machines rely on compiler to utilize ILP. This helps in the predictability of the software because the hardware does minimum reordering or stalling. Yan et al. [103] Studied the predictability of VLIW machines, and proposed changes to the architecture and compiler to improve the predictability. Although most of the data dependency is scheduled away by the compiler, there are still several factors that limit its predictability on the hardware:

- Memory access latency will still cause instructions to stall from the architecture. Since statically it is not known whether a memory access is a hit or a miss, the hardware still needs to check and stall for it.
- Data dependency across compilation units. The hardware still supports basic data dependency checking because across compilation units there could still be dependencies.
- Branch handling. If the VLIW uses branch prediction, there is still the need for handling of mis-prediction etc.

To circumvent the problem, [103] proposed two major methods. First, by using the predicate instructions and full if-conversion combined with hyperblock scheduling they eliminate all none-loop branches. Then, for the dependencies across compilation units, they use code padding. They detect the leaf instructions (instructions which the compiler didn't detect any dependencies) and then based upon how many cycles the instruction takes, and how many c cycles are already there, they

pad the code with nops. This will enable easier WCET analysis. This work currently doesn't deal with caches however, as it assumes a perfect cache.

5.1.4 Real Time Scheduling with Multithreading

With multithreading done explicitly in hardware, the scheduling policy is key to obtaining predictability for multithreaded processors. If the scheduling policy is done by the hardware, and it's not transparent to the programmer, than there is no way to guarantee performance of any thread because the hardware can swap threads without knowledge of programmer.

Kreuzinger et al. [47] (Theo Ungerer) evaluated using different real time scheduling schemes to schedule hardware threads and handle external events. They evaluated FPP (fixed priority preemptive), EDF (earliest deadline first), LLF (least laxity first) and GP (guaranteed percentage), which is during a period, each thread gets a fixed percentage of the pipeline share. The architecture used for this work is a java multi-threaded superscalar pipeline with four threads. [46] explains the architecture in more detail. A hardware priority manager is implemented to facilitate the scheduling of threads. All real-time threads registers its real-time requirements during initialization stage to the priority manager. When the external event occurs, the priority manager schedules the corresponding IST (interrupt service thread) and starts assigning priorities based upon the real time requirements. The evaluation criteria to compare scheduling policies is the throughput of the processor. The conclusion of the report is that in order to maximize multiple threads on a superscalar machine, the scheduler should try and keep as many threads active as long as possible to leverage thread level parallelism and hide more latencies of pipeline stalls. Thus GP does the best because it schedules different active threads each cycle until their percentage runs out, thus it keeps threads alive as long as possible. The idea of using hardware threads to service interrupts is novel because of the low overhead to switch contexts. Also by giving the interrupt service routine thread priorities, you can bound the time. But this approach lacks composability between the different priorities.

A more static approach was proposed by El-Haj-Mahmoud et al. [27] called Virtual Multiprocessor. The idea of a virtual processor is a slice of time on the process. This approached used a multithreaded superscalar in order pipeline, but instead of using a dynamic real time scheduler in the architecture, it attempts to statically schedule the different threads in the architecture offline. The different ways of the superscalar is partitioned and separated, which can be used by the static scheduler to execute threads. It uses scratchpad memory and static branch prediction (pad branch penalties in WCET analysis). For the superscalar, it first introduces a fetch buffer between the scratchpad and issue logic. This is to buffer instructions for all (four in the paper) threads so there is always instruction available to execute. The fetch buffer is filled according to the static schedule, for example, if only one way is reserved for a thread, it is only fetched once per fetch round (every four cycles in this paper). The decoding and scoreboarding logic is duplicated for different hardware thread spaces to check dependencies and hazards within the thread. Also, when a reconfiguration of the processor partition occurs, there might be instructions still in the issue and decoding logic that are waiting to be dispatched from a thread that is going to be disabled. If this instruction stalls, it might create interference in the static schedule because it could block the newly activated threads from executing. Thus, they introduce shadow buffers (one per VP) to store and checkpoint instructions from a disabled hardware thread, so that it could be resumed when it is scheduled. It's unclear in this paper how multicycle instructions could effect the execution of other threads. Also, it makes no mention of how scratchpads interact with main memory, since each VP has its own scratchpad,

it's possible that there is contention to the memory.

Both these multithreaded archtectures extend a superscalar processor for performance reasons. It's unclear really how precise the WCET could be, since a superscalar pipeline itself is hard to obtain precise WCET, accounting in different interrupts and priorities or different threads could only give a really conservative WCET estimate. Even if we try to statically schedule the hardware threads, the schedulability test done needs to be based on a conservative assumption of the WCET. The question is, how precise is the WCET of a superscalar without the presence of caches and branch predictors.

Virtual Multiprocessor: An Analyzable, High-Performance Microarchitecture for Real-Time Computing [27]

- What is the background of this work? What is the motivation?
  High-end embedded systems demand analyzable high performance. To meet higher performance targets features like deep pipelining, dynamic branch prediction, multiple instruction issue, multithreading, out-of-order execution have been introduced in embedded processors. Deriving tight and safe bounds on the WCETs of tasks on such processors is intractable. Higher performance can be achieved without sacrificing analyzability by using multiple simple processors. However, the uniform partitioning of resources among multiple processors leads to load-balance problems. In Simultaneous Multithreading (SMT), resources can be shared more flexibly to better utilize aggregate resources. However, as simultaneous tasks interfere in SMT, this flexibility comes at the cost of reduced analyzability.

- What is the main goal?
  The goal is to achieve high performance (and therefore increased schedulability) without sacrificing analyzability.

- What did they do to achieve this goal? They introduce a Real-Time Virtual Multiprocessor (RVMP). RVMP is an extension of an in-order superscalar processor. The idea is to virtually partition the superscalar pipeline in *space* and *time*. In the space dimension, the processing resources of the processor may be partitioned arbitrarily. If the underlying superscalar architecture has four ways, it could be partitioned into four virtual processors with one way, a single virtual processor with four ways, or anything in between, like one virtual processor with three ways and one with one way. Over time, these partitions may change. The partitioning is done in such a way that tasks executing on the virtual processors do not interfere, i.e. they achieve timing composability. This enables independent analysis of tasks on analyzable virtual processors. Interference on memory accesses is eliminated by the introduction of scratchpad memories.
  For scheduling, time is split into *rounds* of fixed length. Each task is split into subtasks that are executed with these rounds. Scheduling tasks in rounds has the benefit of yielding more compact schedules which can be stored efficiently in the processor. The impact of splitting tasks on their schedulability is unclear.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work? The approach is evaluated by performing schedulability analyses. Benchmarks from the C-lab real-time benchmark suite and the MiBench embedded benchmark suite are used to

generate random task sets with four (eight) tasks. The periods of the tasks in the task set are randomly selected with some constraints to eliminate trivially schedulable and unschedulable instances. The schedulability of these task sets is then evaluated on RVMP and on fixed partitions of the processor: 4x1, 2x2, and 1x4. Task sets with high utilization can be scheduled on RVMP significantly more often than on processors with fixed partitions. In a second set of experiments, RVMP is compared with two SMT architectures. As these SMT architectures are not analyzable, schedulability is determined using simulation, which is unsafe. In these experiments, RVMP achieves similar performance as the two SMT architectures.

- What other work is listed as future work? No future work is listed.

Additional questions:

- What are the limitations/assumptions of this work?
  It is unclear how accesses to shared resources, in particular main memory are dealt with. It appears that all memory accesses go to the scratchpads, which is unrealistic. The paper assumes a simple task model with periodic tasks, where each task's deadline is its period. As instruction fetchs of different virtual processors are performed sequentially, it seems that the order of instruction fetches would have an influence on the WCET of a task. This influence seems to be ignored. The effect of scheduling tasks in rounds on schedulability is not discussed.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...) Hardware: the entire superscalar pipeline, scratchpad memories instead of caches.
  The existence of WCET analyses for virtual processors is assumed.
  A new scheduling mechanism needs to be employed.
  The use of scratchpads entails the use of a compiler to perform allocation or manual allocation.

Virtual Simple Architecture (VISA): Exceeding the Complexity Limit in Safe Real-Time Systems [8]

- What is the background of this work? What is the motivation?
  Modern processors include feature like deep pipelining, dynamic branch prediction, multiple instruction issue, multithreading, out-of-order execution. These features increase average-case performance and energy-efficiency. However, deriving tight and safe bounds on the WCETs of tasks on such processors is intractable.

- What is the main goal?
  The goal is to profit from architectural advances, which are intractable for WCET analysis, but improve throughput and energy consumption, without sacrificing safety.

- What did they do to achieve this goal?
  They propose virtual simple architectures (VISA). A VISA specifies the timing of a hypothetical simple pipeline that is amenable to safe and tight WCET analysis. A microarchitecture should have a simple mode in which it conforms to the VISA specification. Outside of this safe mode, it can use arbitrary performance-enhancing features. Splitting tasks into subtasks

allows to profit from the advantages of the modern processor in the average case: Tasks are speculatively executed in high-performance mode, as long as execution time is within WCET bounds determined under the VISA specification. Only if this is "close to become unsafe" is execution switched to safe mode, which is then still able to meet the deadline. In most executions, this will not happen, and one can benefit from the architectural improvements in terms of energy efficiency and performance. The paper specifically studies the use of dynamic voltage scaling (DVS): Because tasks are typically executed much faster in high-performance mode the operating frequency can be reduced to safe energy.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They use six benchmarks from the C-lab real-time benchmark suite. These benchmarks are then split into five to ten subtasks. Simulated execution times in high-performance mode are between three and six times higher than in safe mode. Accordingly, the processor frequency can be significantly reduced by reducing frequency to 125 to 225 MHz in high-performance mode as opposed to 375 to 600 MHz in safe mode. This yields energy savings between 10% and 70%.

- What other work is listed as future work?
  Instead of using slack to save energy, one could execute other soft-real-time and non-real-time tasks.
  Another direction would be to investigate methods for ensuring VISA-compliance that do not require a simple mode of operation, but rather comply by design.

Additional questions:

- What are the limitations/assumptions of this work?
  Because of overheads due to switching between high performance and safe mode and a limited number of checkpoints, there has to be some slack to begin with. In other words, a system that is barely schedulable according to the WCET bounds cannot profit from the described approach.
  A "real" safe architecture may be run at a slightly higher clock frequency[1], than the safe mode of a high-performance architecture, rendering some task sets unschedulable on the high-performance architecture, that would have been schedulable on a "real" safe architecture.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...)
  Hardware: entire system. Task execution has to be split up into sub-tasks.

5.1.5 Real Time SMT

Several work involved trying to utilize SMT for real time systems. Barre et al.[14] (Pascal Sainrat) took this idea of Simultaneous Multi-Threading and pushed the idea for real time systems. This seemed to be counter intuitive at first, since the dynamic scheduling of threads and sharing of resources would be a nightmare for worst case execution time analysis. Motivation for this work

[1] The paper assumes 50%.

include the "Integrated Modular Avionics (IMA)" movement [71], which tries to integrate several tasks into a single computing node. This can save power (on interconnects/communication) and save weight for air crafts. However, for the current paper[14] caches and branch prediction are listed as future work, which are the key aspects to it. The idea is very simple, give one explicit hardware thread the highest priority, and it gets access to any resource whenever it needs to schedule it. The hardware *preempts* any lower priority thread instruction that's using a resource for the highest priority thread. It later replays the instruction. Along with the scheduling, any resource that needs to be shared (like instruction queue or decoding queue) is *partitioned* to reduce interference. This gives the highest priority thread the illusion that it has the whole core to itself, while other threads are scheduled when resources are free from the high priority thread. The same concept was also proposed earlier by Uhrig et al. [76] (Theo Ungerer), except for in order instead of out of order execution. As this not only helps WCET analysis, but also Hily et al. [36] proved that out of order may not be as cost effective as in order on SMT machines.

Mische et al.[57] (Theo Ungerer) expanded this work and came up with a way to have more than one real time thread running in the architecture. Along with having one priority thread that gets resource any time it needs it, you time share that thread in a coarse grained way for different real time threads (a static schedule is constructed). The real time threads only execute in the high priority slot, so it can have timing guarantees and analyzability. This paper assumed one instruction scratchpad without a data scratchpad, and no branches (although it argues that the lack of branches can help the non real time thread's throughput). This paper notices some problems with memory access, as you can't partition the memory access between threads. Two problems arise, first is when a time shared high priority thread needs to access memory, but a low priority thread already has outstanding memory request in flight. The proposed solution is to alert the memory controller as early as possible when a high priority thread has decoded a memory instruction. This way the memory controller will hold off on all requests that are queued, and wait to service the high priority thread. This reduces the additional latency penalty by the number of pipeline stages between issue stage and memory controller. However, the memory latency is usually in the hundreds of cycles, thus it's not very plausible (in this paper they only had a data memory access of 3 cycles, so they claim to have completely hidden the memory latency of low priority threads). The second problem is during the context switching of real time threads, if the last instruction of the previous context is a memory operation, and the next context needs memory right away, there is a contention problem between the real time threads. This paper doesn't directly deal with the problem, but simply extends the time allotted to a thread, and decreases its time during the next round. For analysis, it seems it needs to conservatively assume that this might happen every time. Although this penalty will never aggregate, but the penalty could be bad with a high number of real time threads and long memory access latencies.

A Predictable Simultaneous Multithreading Scheme for Hard Real-Time [14]

- What is the background of this work? What is the motivation?
  Complexity in embedded software has grown, but increasing cores might not be an ideal solution because the complexity of interconnect. So SMT could be a good solution. But currently SMTs do not provide timing predictability at all.

- What is the main goal?

The main goal is to provide a **WCET-aware** SMT processor to execute one real time thread in parallel with less critical threads. In other words, to design a SMT architecture that makes it possible to analyze WCET of real time tasks. The HRT thread should have no interference with other threads.

- What did they do to achieve this goal?
  They identified pipeline resources that are storage resources (instruction queues and buffers) and bandwidth resources (functional units and commit stage). The storage resources are statically partitioned among threads to reduce interference. The instruction fetch is S-RR policy (every cycle a different thread is fetched regardless if the thread is ready) to the fetch queue. Then a Most-critical-first scheduling policy is implemented to choose instruction to decode from the fetch queue. Basically, the HRT thread is always chosen first, then the non real time threads get chosen. For the functional units, if the HRT thread needs the a functional unit and it is currently occupied by a NRT thread, then the NRT thread gets trashed and later replayed so the HRT gets access right away. This way the HRT does not have any interference from the NRT thread.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  A cycle-level simulator is used to simulate the processor architecture. They used a perfect branch predictor and perfect data cache, and without modeling the instruction cache, they used a random 1% instruction miss rate with 100-cycle miss latency. They compared the WCET-aware architecture (mostly MCF policy) with a baseline core that implements O-RR scheduling policy for all resources and oldest first scheduling for functional units etc.

  They use some simpler functions from the SNU-RT suite and run the same function on all threads to maximize interference. First they run the function with dummy threads to get the performance of the function without interference. Then they begin to run the function on all threads and they show the performance of the last finishing thread. They state that the predictable SMT has no change in execution time compared to the dummy thread experiment (although it's not shown in numbers or tables), and they show that their approach has performance degradation compared to the baseline core, in which the last thread to finish takes longer. They call it a "moderate degradation". No formal analysis or WCET analysis is shown.

- What other work is listed as future work?
  Address some issues that were ignored in this preliminary study, like the strategy for sharing the instruction and data caches, as well as the branch predictor, so as to maintain full timing predictability for the critical thread. We will also investigate solutions that would allow the concurrent execution of several critical threads.

Additional questions:

- What are the limitations/assumptions of this work?
  They assume perfect caches and branch prediction, and mainly focus on pipeline features.

  They assume it's possible to do WCET analysis of out of order superscalar processors

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  The hardware is modified by this approach, in particular, mechanisms to partition shared queues and trash and replay instructions in the functional units are added to support implementing priorities in an SMT architecture.

Exploiting Spare Resources of In-order SMT Processors Executing Hard Real-time Threads [57]

- What is the background of this work? What is the motivation?
  SMT processors have higher utilization of processor resources and higher throughput. So the authors want to leverage that higher utilization and throughput but still allow static WCET analysis. (note that there is no clear definition of how precise the static WCET analysis methods are, or how difficult they could be). This work also attempts to do so in the presence of multiple real time threads.

- What is the main goal?
  The paper makes the following four contributions:

  - An SMT architecture that allows a static WCET analysis
  - A scheduling algorithm that executes multiple HRT threads concurrently in SMT processor
  - An issue policy that uses free resources for NRT threads without interference
  - Solutions to handle multicycle memory access

- What did they do to achieve this goal?
  An In order execution policy is used, as compared to [14]. This helps in static analysis as well as improved throughput [36] compared to out of order execution. Although the paper doesn't go into specific details about the architectural features that were changed (beyond the basic single thread to multi-thread hardware conversion), it does specifically state that any changes they had made to the architecture ensures that the HRT thread executes as if it were executing alone.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  Benchmarks are run from the EEMBC automotive benchmark suite and the Malardalen WCET group. They were executed on a cycle-accurate SystemC model of CarCore. The first benchmark they looked at was the effects of the memory instruction announcing. They looked at the performance of the HRT when there was only 1 instruction or no instruction announcing (CarCore assumes 2 instruction announcing can completely hide the other threads from the HRT thread). They showed that there was a slight performance degradation for the HRT thread, meaning that there was interference from the other threads. They noted that this performance degradation however mainly depends on how much memory access is done by the HRT thread. On the flip side, they also showed that the NRT threads gain in performance (compared to the 2 instruction announcing) when there is 1 or no instruction announcing. All

graphs were shown in relative percentage. The second set of numbers they showed the utilization of the processor. They showed that within the HRT thread, if there is only one task, then the utilization is 100%, if there are 2, then they are both 50%. This shows that there is no interference from NRT threads. Note that NRT thread utilization drastically decreases as priority lowers.

- What other work is listed as future work?

Additional questions:

- What are the limitations/assumptions of this work?
  One assumption is that program execution on superscalar in-order processor is deterministic, so static WCET analysis is possible (again, nothing about its precision etc).
- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)

Metzlaff et al. [55] (also Theo Ungerer) also proposed a method to deal with instruction caching. First they propose a separate scratchpad for each thread. Then they used method cache [42] with scratchpads and give priority to the top thread when a filling is needed. If the thread is stalled when a method is being filled into the scratchpad, then it seems there would be no data cache access conflict. By using a SMT machine, if the high priority thread is filling its cache, the lower priority threads could still fill the pipeline and gain utilization and throughput.

Although real time SMT proposals have a lot of compelling features, it still lacks a formal WCET analysis. All of the proposals have just used measured benchmarks to prove performance, and assumed that because the high priority thread runs as if no other thread is running, so static analysis is possible. In reality, WCET for a superscalar processor already contains pessimistic results. Although the lack of a branch predictor and in order execution may aid in this. Also, in this line of work still only one explicit hardware thread slot gets real-time performance, while the other threads' performance will have a non-continuous penalty, which is hard to categorize.

5.1.6 Real Time Java

Schoeberl designed the Java Optimized Processor (JOP) [80] to propose using Java for real time embedded systems. The design of JOP includes a two level stack cache architecture [79]. Instead of using a large register file to store the stack like in PicoJava[53], it only uses two registers to store the top two entries of the stack (Register A and Regiser B). Leveraging the stack based architecture of JavaVM, whenever an arithmetic operation occurs, the result is always stored back to the top of the stack (Register A). Any push or pop operation simply results in a shift of values between the two registers and the stack cache, which only requires one read and one write port for the memory. This architecture does 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. Also, all bytecode on JOP is translated into a fixed length microcode.

Each microcode executes in a fixed amount of cycles (mostly 1 cycle) independent of its surrounding instruction, thus the WCET analysis only requires a lookup table of bytecode translated into microcode, and it knows the execution time.

5.1.7 Thread Interleaving

Old idea: HEP[83], Lee and Messerschmitt[51],
Industrial use: XMOS, Ubicom IP3023, Sandbridge Sandblaster, Infineon Tri-Core

5.1.8 Missing papers

- PRET-C - Partha Roop
- MCGREP - Jack Witham

5.2 Memory Hierarchy

5.2.1 Caches

The basic block of caches is a cache line. A cache line can contain B words. When the cache is filled, the whole line is filled. When a replacement is needed, the whole line is replaced. The cache can be N-way associative, where N is the number of lines in each set. If a cache is 1-way associative(direct mapped) with size S, then it has $M = S \div B$ sets, where M is the number of sets in the cache. A cache size therefore is determined by $S = M \times N \times B$.

If a cache is direct mapped, it does not need replacement policy, since you always know which line needs to be evicted. However, if the cache is more than one way, then a replacement policy is needed to determine which way is to be replaced.

Talk about tradeoff in cache size/cache way etc

Unified vs Separate Cache

Typically a cache of an architecture is structured in 2 ways. A *Unified Cache* is one where instruction and data are in the same cache. A *Separated Cache* is one where instruction and data are on separate caches, each with independent state information. A unified cache introduces interference between instruction and data accesses. Thus, this complicates the WCET analysis even more because data caches are harder to model or determine the value. Thus, Separate caches for instruction and data should be used to help obtain a more precise and simpler analysis.

Replacement Policies

To read: [35]

Timing Predictability of Cache Replacement Policies [73]

- What is the background of this work? What is the motivation?
  It had been observed in previous work, e.g. [35] that cache analyses for LRU (least-recently-used) were more precise than analyses for other policies like FIFO and PLRU. However, it was not clear, whether this was because analyses of FIFO and PLRU were less developed than those of LRU, or whether LRU was simply more "predictable" than other policies, and analyses of similar precision were unattainable for FIFO and PLRU.

- What is the main goal?
  To study the predictability of cache replacement policies and to explain empirical observations.

- What did they do to achieve this goal?
  Cache analyses have to cope with uncertainty. Sources of uncertainty are, among others:

  – The cache contents when a task is started are usually unknown.
  – If the address of a data accesses cannot be precisely determined, this introduces uncertainty about the cache state.
  – Preempting tasks may change the cache state at preemption points.

  These sources of uncertainty are independent of the particular cache organization. The precision of cache analyses is therefore determined by the speed at which information about the cache state is obtained. They defined two predictability metrics that determine how quickly knowledge about cache hits and misses can be (re-)obtained for a particular cache replacement policy. These metrics are independent of any particular cache analysis, i.e. they mark the precision limit for any possible cache analysis. They go on to evaluate the policies LRU, FIFO, PLRU, and MRU under these metrics. The values of the different policies confirm the expectation that LRU is significantly more predictable than other policies. It, however, also revealed potential for improvement in existing analyses of PLRU and in particular FIFO.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They compare their results to previous empirical evidence in cache analysis, which was based on specific static cache analyses.

- What other work is listed as future work?
  To investigate the precision of cache analyses and to develop new cache analyses that are optimal w.r.t. the metrics (partly solved in subsequent work).

Additional questions:

- What are the limitations/assumptions of this work?
  A limitation of the metrics is that they assume complete uncertainty, whereas partial information about the cache state is often available.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  This work does not propose a new feature. However, it recommends the use of LRU replacement in caches, which affects the cache itself and the WCET analysis.

Instruction Cache

A Time Predictable Instruction Cache for a Java Processor [78] Schoeberl [78] was the first to propose Method Caches (Called function cache in[55]) for the instruction cache. The idea of a method cache is that the granularity of the cache block replacement size is now not a cache block, but a method instead. When a method is called, the cache content is loaded with that method. The cache could contain a single block, two block, or variable blocks to load the caches. The different block size gives different trade offs. The single block gives perfect predictability, but bad performance, since every time a method is called or returned to, it needs to be loaded. Not only that, some methods are loaded only to execute a few instructions before calling another method, resulting in high overhead. The two block method cache can cache two methods. The replacement policy would be LRU. This gives better performance, because on a return to the caller method, the cache will yield a hit (if no other methods were called). The performance of the two block is about two times better than a single block. To further improve, the cache could be split into several blocks. A method could have the size of several blocks, but some methods may only need one block. When a method is loaded, it occupies the amount of blocks it needs. When a method is evicted, all blocks it occupies are evicted. A pointer stores the next block to replace or load methods in. This could give better performance because depending on the block size you chose and the method sizes, you could more efficiently using the cache blocks. But if there are a lot of blocks, it could also increase the WCET difficulty.

- What is the background of this work? What is the motivation?
  Standard cache organizations improve the average-case execution time but are difficult to predict for WCET analysis. (JR: Unfortunately, this is really the only motivational part I could find in the paper.)

- What is the main goal?
  A cache organization that is amenable to simpler and more accurate WCET analysis. Performance should not be completely neglected.

- What did they do to achieve this goal?
  Schoeberl proposes the method cache. Instead of caching memory blocks of fixed size as in a traditional cache organization, the method cache caches entire methods. This simplifies cache analysis as there are fewer points in a program at which the cache is filled: this may only happen on calls and returns. It is claimed that tag memory and address translation are not necessary in a method cache. (JR: Something like a tag memory is still necessary to remember which methods are cached.) Several variants of the method cache are introduced: single method cache, two block cache, variable block cache. The single method cache always caches the current method and nothing else. The two block (why not method?) cache can cache up two methods. It can easily be implemented with LRU replacement. To reduce the waste of cache memory the variable block cache is introduced, which can store more than two methods. The variable block cache can store a fixed number of blocks, which are similar to cache lines in traditional designs. In this organization, LRU replacement cannot be realized. Two replacement policies are proposed: next block (FIFO) and stack oriented. WCET analysis is briefly sketched.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  The new cache design is evaluated by simulations of a single application from one node of a distributed motor control system. For one cache size (2KB) and the variable block cache with 32 blocks, the method cache outperforms a direct-mapped cache of equal size. The improvement in predictability is not quantitatively evaluated. It is argued, that the single method cache is more predictable than the two-block cache which is in turn more predictable than the variable block cache. WCET analysis of the variable block is "nevertheless simpler than that with the direct-mapped cache". Filling the cache only on method invocations and returns removes potential competition with data accesses.
- What other work is listed as future work?

Additional questions:

- What are the limitations/assumptions of this work?
  It deals with instruction caches only.
- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...)
  The java processor and the WCET analysis.

Static Analysis of the Method Cache [42]

- What is the background of this work? What is the motivation?
  The method cache was introduced as a time-predictable instruction cache. However, only simple variants had been statically analyzed before.
- What is the main goal?
  To develop a precise static analysis of the method cache.
- What did they do to achieve this goal?
  They adapted the techniques of Ferdinand et al. to method caches and introduced the notion of local persistence analysis. Local persistence analyses determine whether a memory reference within a scope (therefore local) can cause more one cache miss.
- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  The new analyses is evaluated on a small synthetic benchmark consisting of "a few functions, two loops and a [sic] if statement". They compare the WCET bound obtained with the proposed analysis to the WCET bounds obtained with two previous analyses and a measured execution time.
- What other work is listed as future work?
  To finish the implementation of the analysis and to compare it to existing analyses of set-associative caches on a collection of benchmarks.

Additional questions:

- What are the limitations/assumptions of this work?
  According to Martin Schoeberl, this analysis is incorrect. The abstract updates are not described in the paper. With the given domains, sound updates should not yield any "non-trivial" classifications.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...)
  WCET analysis.

Trace Cache

Static Cache Locking

Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems [69]

- What is the background of this work? What is the motivation?
  Schedulability analysis depends on safe bounds on the WCETs of tasks. Caches have a strong influence on the execution times of tasks. It is hard to statically predict execution times in the presence of caches: Such an analysis has to consider both intra- and inter-task interferences. Intra-task interferences occur when different memory blocks of the same task compete for cache blocks. Inter-task interferences, imply a so-called cache-related preemption delay, where a preempting task's memory blocks cause cache reloads in the preempted task. Analyses for both intra- and inter-task interferences have been developed. An alternative to analyzing these interferences is eliminating them. Task/cache partitioning and cache locking have been proposed for this purpose. In cache partitioning a part of the cache is reserved for a particular task. This eliminates inter-task interferences, but intra-task interferences remain. Similarly, the data layout of tasks can be adapted in such a way that different tasks do not interfere in the cache. This is known as task partitioning. Finally, cache locking has been proposed to eliminate both inter- and intra-task interferences. In cache locking, the contents of the cache are loaded and locked at fixed times. In *static cache locking* they are fixed at system start for the whole system lifetime, whereas the contents may be changed in *dynamic cache locking*, for instance at preemptions. As what is being locked into the cache is decided statically, it is usually [2] easy to predict the memory access times.

- What is the main goal?
  To explore the use of static cache locking of instruction caches in multitasking real-time systems.

- What did they do to achieve this goal?
  They propose two algorithms for static cache locking. The first algorithm minimizes the utilization of a task set. Typically, lower utilization results in better schedulability. Sufficient (and in some cases necessary) schedulability conditions based on the utilization exist for different scheduling regimes. For fixed-priority scheduling, necessary and sufficient response-time analysis (RTA) methods have been developed. The second proposed algorithm tries to

[2] Cache locking techniques could, in principle, mimic the dynamic behavior of a cache.

increase schedulability by minimizing interference between different tasks, which is considered in RTA.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They compare their two static cache locking approaches with a regular unlocked cache. They analyze the WCET in case of an unlocked cache using a (at the time) state of the art instruction cache analysis [59]. To estimate the cache-related preemption delay (CRPD) they assume that all blocks of a task have to be reloaded after a context switch (At the time a more accurate analysis of CRPD was available: Lee et al. [49]. More precise analyses based on the original approach have been developed recently). They analyze two task sets, each consisting of four tasks. For these tasks, they choose a period that would result in a utilization of 1.3 if no cache would be employed. For a number of cache configurations of varying associativity and size, they report the utilization determined by static analysis. In addition to reporting utilization, they also indicate whether the task set was deemed schedulable by schedulability analysis. For small associativities, the unlocked cache significantly outperforms statically locked caches in terms of utilization and schedulability. For higher associativities, the static cache locking techniques perform better, as they are more flexible in their decisions of which instructions to lock. On the other hand, the static analysis of the unlocked cache is less precise for higher associativities[3]. For associativities greater than 4 or 8, static cache locking outperforms static analysis of unlocked caches.

- What other work is listed as future work?
  To investigate average-case performance of static cache locking. The proposed algorithms assume a fixed "worst-case execution path". In future work, they would like to investigate how sensitive the algorithms are to this path. They would like to compare their algorithms to genetic algorithms proposed by Campoy et al. For large programs, they plan to explore dynamic cache locking strategies.

Additional questions:

- What are the limitations/assumptions of this work?
  The cache supports locking contents.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...)
  Compiler, WCET analysis.

5.2.2 Scratchpads

Scratchpads (ARM refers to this as Tightly Coupled Memory - TCM) are a form of fast access memory, except the allocation of the data on the scratchpad is controlled by software, where as caches are controlled in hardware. This is done often by mapping a memory space as the scratchpad space, where all accesses to that space will go to scratchpad and all other accesses will go to memory. This allows more precise WCET analysis [94] (in the absence of caches) because based

[3] This is surprising and not explained.

upon what memory location is being accessed, you can categorize the access latency (of course one difficulty in WCET is knowing what the actual memory location is). There are two allocation schemes that are employed by the compiler. A *static allocation scheme* allocates the data before the program runs, and the allocation doesn't change throughout the program (see Static allocation schemes). A *dynamic allocation scheme* can change the mapping of data on the scratchpad during run time in attempt to increase performance (See Dynamic allocation schemes).

Static allocation schemes

Static allocation schemes allocate the content on the scratchpad a priori, and the content stays the same throughout the whole execution. Most static scratchpad allocation schemes (Find some citations) use some sort of heuristic to find the most commonly executed instructions or data structures, and then allocate them statically on the scratchpad to improve the ACET (average case execution time). Suhendra et al. [84] (Abhik Roychoudhury) was the first to propose using static data scratchpad allocation to improve the worst case execution time. They first identified the difficulty in constructing such an algorithm that optimally allocates data to improve the WCET because once data is allocated on the SPM, the worst case execution path will change. Also, it's possible that the WCEP that's resulted from static analysis is actually infeasible, so the detection of infeasible task is also needed. As a result, they proposed a greedy method that allocates contents of the SPM in attempt to reduce the WCET. They first assume that no allocation is done for any memory location. They find the WCEP, and allocate the most used data block (not too sure of what size, does it depend on the data structure? Is it fixed size?) from that path onto the SPM, then reiterates the algorithm to find the new WCEP. This iterates until the SPM space is exhausted. Note that is is sub-optimal because allocated content will not be reconsidered. This means that if the first WCEP and the second WCEP don't share that data structure, the first allocation does not contribute to improving the WCEP.

Patel et al. [65] proposed another static allocation scheme based on [84], except that the allocation criteria wasn't to minimize the WCET, but to meet all deadlines in the program. The greedy approach worked by synthesizing timing constructs (deadline blocks) into test programs. The algorithm works by identifying the deadline blocks that miss their deadline. Then a profile is constructed based on the number of accesses to a data block in the missed deadline blocks. Based on the profile, a data block is selected to be allocate on the scratchpad, and then the algorithm reiterates itself, until either all deadlines can be guaranteed to be met, or if the SPM exhausts its space. In the first case, the remaining space can be optimized by another method. In the second case, the program is deemed un-schedulable, and the deadlines cannot be met.

Dynamic allocation schemes

5.2.3 Caches vs. Scratchpad

Puaut et al. [70] did a comparison of locked caches and scratchpads. Some architectures support locking cache lines and software controlled loading of content into the cache. Puaut showed that the difference between using locked caches and scratchpads was minor. Most benchmarks provided similar WCET estimates. The differences stem from the granularity of blocks. For locked caches, the basic allocation unit is a cache line. Thus, it's independent of the basic block size of

the allocation. However, this results in the *pollution* of locked content. *Pollution* occurs when a cache line that's locked contain words that aren't part of the allocation scheme, but simply locked in because it belonged to the same cache line. 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's also locked, and thus lose its ability to be locked in the cache. For scratchpads, the basic allocation is only determined by the allocation scheme. However, if the basic allocation block is big, it's possible that due to the fragmentation, a big allocation block doesn't fit in the scratchpad at the end. In the paper, this resulted in a 30% drop in the on-chip access ratio of total memory accesses.

Scratchpad Memories vs Locked Caches in Hard Real-Time Systems: A Quantitative Comparison [70]

- What is the background of this work? What is the motivation?
  Caches are used to bridge the increasing performance gap between the processor and the off-chip memory. Allocation and deallocation of memory blocks from the cache is managed by hardware in a transparent manner to the programmer and the compiler. In hard real-time systems, caches are a source of unpredictability. A lot of progress has been made to statically predict cache behavior. Due to lack of documentation about the employed replacement policies, this work is not always applicable. In addition, such analyses are relatively pessimistic for some replacement policies, such as pseudo round-robin, pseudo-LRU, or random replacement.
  Many processors allow to lock (also: freeze) the contents of the cache. Static and dynamic cache locking techniques have been developed. An alternative to caches is scratchpad memory (SPM). In contrast to caches, scratchpads are under software-control. The compiler or programmer needs to allocated code and/or data to the scratchpad memory. Significant work has been done to develop allocation techniques for SPMs, most aimed at reducing average-case execution time. Only one previous paper [84] was concerned with WCET-oriented *static* scratchpad allocation.

- What is the main goal?
  To develop a dynamic WCET-oriented instruction scratchpad allocation and cache-locking algorithm. This algorithm should minimize the WCET estimate. The second goal is to quantitatively compare cache locking with scratchpad allocation.

- What did they do to achieve this goal?
  Their scratchpad allocation/cache locking algorithm proceeds in two steps:

  1. Selection of reload points
  2. Selection of on-chip memory contents

  In the first step, a subset of the loop pre-headers is selected as potential reload points. Only at these points, will the algorithm consider to load memory contents into the scratchpad/cache. The second step iteratively and greedily decides which blocks to allocate to the fast memory (scratchpad/cache). To decide which blocks are most beneficial to improve the WCET estimate, WCET analysis is performed. The WCET analysis determines execution frequencies of basic blocks that maximize the execution time of the task. These execution frequencies are

taken into account when deciding which block to allocate to the fast memory. As allocations to the fast memory may change the execution frequencies maximizing execution time, WCET analysis has to be performed again after allocating blocks. The algorithm allows to trade off the number of costly WCET analysis invocations and the quality of the allocation.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  The paper focuses on comparing scratchpad allocation with cache locking. It does not compare the (worst-case) performance of unlocked caches with that of locked caches or scratchpads. The two possibilities are compared regarding the WCET estimates obtained with the Heptane WCET analysis tool. Five benchmarks, four from the MŁlardalen benchmark suite and one from the UTDSP benchmark suite are used. In their initial analysis, the differences between WCET estimates for locked caches and scratchpads are not very large: sometimes locked caches outperform scratchpads, sometimes vice-versa. This is linked to the cache block size and the basic block size. Large cache blocks lead to pollution, where unimportant instructions are locked, as cache blocks can only be locked as a whole. Depending on the alignment of basic blocks with cache blocks, this problem is more or less severe. As the locking of SPM contents is performed at the level of basic blocks, large basic blocks can be problematic: they cause fragmentation, as very large basic blocks might not fit onto the scratchpads. Fragmentation could be a more significant problem with large data structures such as large arrays.

- What other work is listed as future work?

Additional questions:

- What are the limitations/assumptions of this work?
  Splitting basic blocks to reduce fragmentation. Allocation of data.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  Compiler, WCET analysis.

Preemption - Although scratchpads are more predictable, the hardware replacement policy of caches could be advantageous, especially for systems that have preemption of tasks. Locked caches can lock content in the cache, and when a preemption occurs, the hardware can still take advantage of a hardware replacement for lines that aren't locked, while still maintaining a certain amount of predictability when the original task resumes (because of the locked content that aren't replaced). For statically allocated scratchpads, the allocation scheme needs to take into account the preempted task, and allocate space for that task. This will reduce the available allocation space for all other tasks. For dynamically scheduled schemes, it will be extremely difficult to find reload points to adjust the scratchpad. If a preempted task loads its optimal content when it preempts a task, it's unclear what to load back for the original task, and when to load it. There needs to be some sort of OS that keeps track of what was loaded off the scratchpad. It is unclear what overhead this could result in. (Read papers about this (JR: Write paper about how to do this ;-))

5.2.4 DRAM

Predator: A predictable SDRAM memory controller [5, 6, 4]

- What is the background of this work? What is the motivation?
  Contemporary multi-processor SoCs (systems-on-chip) feature a large number of intellectual property components, which communicate through shared memory. Some of these IPs have hard real-time requirements. The memory traffic generated by the different components is dynamic and not fully known at design time. 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

  – the time to service a request depends on past requests,
  – in particular, it depends on past requests by other requestors

  Due to interference on shared resources, verification complexity of real-time requirements increases dramatically on multi-processor systems. If the behavior of different applications can be completely isolated, they can be verified in isolation.

- What is the main goal?
  The first main goal is to provide a memory controller design that provides a guaranteed minimum bandwidth and a maximum latency to each of the requestors (*predictability*) independently of the behavior of other requestors. It should efficiently utilize the memory chip. The second main goal is to provide complete isolation of requestors, i.e. the behavior of one requestor should not influence the service experienced by other requestors (*composability*, this part occurs only in [6, 4]).

- What did they do to achieve this goal?
  The predictability goals is accomplished in two steps:

  – First, a set of of read and write groups, with corresponding static sequences of SDRAM commands, are determined. These groups determine the minimum request size, which can be, for instance, the size of a cache line. Longer groups will increase memory efficiency, as long as no unnecessary data is fetched. On the other hand, longer groups will increase latency.
  – Secondly, a Latency-Rate scheduler (in this case a Credit-Controlled Static-Priority (CSSP) scheduler) is employed to provide the different requestors with the requested service independently of the behavior of the other requestors. CSSP allows to decouple allocated latency from rate, in contrast to TDMA approaches.

  The composability goal is achieved by a special front-end [6, 4]. Essentially, the front-end delays each response by the predictable back-end (memory controller) up to its worst-case bound. It may also reject new requests, if the request queues would be full, assuming worst-case behavior of the memory controller. In this way, interactions between different requestors are completely eliminated.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?

The predictable memory controller Predator is evaluated by simulation of a SystemC model. Four hard real-time requestors are mimicked by traffic generators. These traffic generators have a combined bandwidth requirement of 660 MB/s. In the simulations, the maximum observed delays for the requestors are recorded. These are then compared with the analytical bounds. Reassuringly, the analytical bounds are never exceeded during simulation and all requestors receive their allocated rate. However, for lower priority requestors, the latency bounds are fairly pessimistic as the worst-case is extremely unlikely. In a second experiment, one of the requestors asks for more resources than allocated. The well-behaved requestors are unaffected by this behavior and continue to receive their allocated rate within the analytical latencies.

- What other work is listed as future work?
  Developing an algorithm for automatic generation of memory access groups, given a set of memory timings and a burst size (proposed in [5], done in [4]).

Additional questions:

- What are the limitations/assumptions of this work?
  It seems that the guaranteed latency is rather high. Guaranteed bandwidth is close to the maximum that can be provided. However, having to account for potential refreshes and performing rather long bursts (through all banks), results in long worst-case latencies, in particular for lower priority tasks, as noted in [63]. This can be partly alleviated by a different arbiter, like round-robin or TDMA, which in turn has other disadvantages.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  WCET analysis is simplified.

An Analyzable Memory Controller for Hard Real-Time CMPs [63]

- What is the background of this work? What is the motivation?
  Multicore processors provide the performance required by current and future hard real-time systems. However, due to interferences on shared resources, it is difficult to compute tight bounds on the WCETs of tasks running on such processors.

- What is the main goal? An analyzable JEDEC-compliant DDRx SDRAM memory controller for hard real-time multicores. The WCET estimation of a task should be independent of the memory behavior of other corunning tasks (composability). Furthermore, the controller and corresponding analysis should be adaptable to arbitrary JEDEC-compliant DDRx SDRAM devices.

- What did they do to achieve this goal?
  They schedule memory accesses hard real-time tasks (HRTs) in a round-robin fashion. These accesses are prioritized over those of non hard real-time tasks (NHRTs). Memory requests are serviced in bursts, which are interleaved over all banks. (JR: Note, that this approach interleaves accesses belonging to the same request as opposed to our PRET approach.) They analyze the maximal delay a memory access can suffer due to a previous access using generic

DDR timing constraints. Based on this delay they compute the maximum access delay taking into account the round-robin scheduling. The resulting "upper bound delay" (UBD) can be used in WCET analysis. They do not change the auto-refresh mechanism. They propose synchronizing the start of a HRT with the the occurrence of a refresh operation at analysis and runtime. (JR: How much this helps w.r.t. to WCET analysis is questionable.)

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They use a hard real-time application from Honeywell, a collision avoidance algorithm to evaluate their approach quantitatively. They consider four scenarios: WCET mode i, with $i \in \{1, 2, 3, 4\}$. In WCET mode i, there are $i - 1$ corunning HRTs and one NHRT. For each of these modes they perform simulations (MOET) and a WCET analysis using the UBDs (AMC). They also perform a WCET analysis assuming a private SDRAM for each task (PRMC). In both cases, the WCET analyses account for interferences on on-chip resources. They also vary the size of the caches from 128KB to 8KB. The smaller, the cache, the more memory requests will be generated by the HRTs. The NHRT is configured to constantly access the memory and to thereby interfere in the strongest possible way. In WCET mode 1, i.e. the HRT competes with no other HRTs but one NHRT, the three analyses (MOET, AMC, PRMC) yield very similar results. In WCET mode 4 and 8KB cache, measurements (MOET) show a slowdown compared with PRMC of around 1.3. AMC computes a bound which is about 1.7 times higher than PRMC, indicating an overestimation around 1.3.

  They compare their approach with Predator [5]. The first argument in favor of their approach is its applicability to arbitrary JEDEC-compliant DRAM devices. However, this requirement was also lifted in Akesson's PhD thesis. They also compare the UBDs provided by Predator with those provided by their own approach. In Predator each HRT is assigned a priority, with decreasing priority, the UBDs rise. While the UBD of the highest priority task is slightly better than the UBD of AMC (their approach), the guarantees for lower priority tasks are much worse in Predator. This is caused purely by Predator's backend, which could be easily replaced by any Latency-Rate server, such as the one presented in this work.

- What other work is listed as future work?

Additional questions:

- What are the limitations/assumptions of this work?
- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...)

Making DRAM Refresh Predictable [17]

- What is the background of this work? What is the motivation?
  DRAM cells leak charge and have to be refreshed periodically to retain their state. This is usually done through auto-refreshes issued by the DRAM controller. From a task's perspective such auto-refreshes occur asynchronously. Refreshes affect timing of memory accesses in two ways:

  - by stalling DRAM accesses during refreshes,
  - by closing DRAM rows, which leads to additional precharges to reopen them.

  Therefore they are difficult to account for in WCET analysis.

- What is the main goal?
  Eliminate interferences between refreshes and memory accesses of tasks, such that WCET analysis can be performed without considering refreshes.

- What did they do to achieve this goal? They execute refreshes in bursts. These refresh bursts are scheduled in periodic tasks. This way they can be taken into account during schedulability analysis. They present two implementations of bursty refreshes. One purely software-based, the other relying on the auto-refresh capabilities of the DRAM controller.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They evaluate their approach by measurements on a DSP and an ARM platform. To analyze the effect of refreshes on execution times they deactivate the instruction and data caches. This way every memory access goes to the DRAM. With standard DRAM auto-refresh they observe a small jitter of less than 0.1% in the execution times of the bubble sort algorithm. Their approach eliminates this jitter completely and improves execution times slightly, by 0.18% to 2.8%. The overhead incurred by the two bursty refresh mechanisms is between 3 and 16%. They also observe, using measurements of the energy consumption of the DRAM module, savings of about 5%. However, it should be noted, that the overhead incurred by the refresh tasks is not taken into account in these savings. They will likely be greater than the savings. The savings will also be reduced by the use of scratchpads or caches.

- What other work is listed as future work?
  They would like to pursue FPGA-based motifications to the DRAM controller to add native support for burst refreshes in hardware. Burst refreshes could be overlaid with non-memory based activities.

Additional questions:

- What are the limitations/assumptions of this work?
  This work considers unpredictability of DRAM accesses due to refreshes. It does not consider variations in access times due to the access history or due to competing requestors. These variations are somewhat different in nature as they can in principle be analyzed in the WCET analysis of a task.

- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  Scheduling, possibly the DRAM controller

Implementing Time-Predictable Load and Store Operations [97]

- Basic definitions:
  Pointer aliasing: The same memory location may be referenced using different names (pointers). **Pointer invalidation:** 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. **Whole-program Pointer analysis:** Determine which pointers can point to which variables and storage locations in the entire program.

- What is the background of this work? What is the motivation?
  Scratchpad (SPM) memories provide time-predictable accesses for data. However, the time-predictable SPM allocation strategies for data only support statically allocated data or stack data. Dynamic data, on the other hand, is only supported by non time-predictable allocation schemes if whole-program pointer analysis identifies every memory operation that could access each variable.

- What is the main goal?
  The objective of this paper is to implement a scratchpad memory management unit that transfers data between external memory and scratchpad memories such that pointer aliasing and pointer invalidation are eliminated. This approach lifts some of the restrictions (i.e. eliminate pointers entirely from program code or no support for dynamic data) forced by the limitations of WCET analysis.

- What did they do to achieve this goal?
  They implemented a scratchpad memory management unit (SPMMU) in hardware that performs a mapping from logical addresses (used by the program) to physical addresses (identifying where an object resides). In addition, the SPMMU performed DMA transfers between the external memory and SPM. The program dictates when the transfers take place 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 but updates an internal table mapping the logical address to the new physical location of the object.

- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They use three approaches to evaluate their work. Their first approach simply looks at the hardware cost incurred. In particular, they observe that since the mapping must take place in the same cycle that the request is made, it must be implemented as combinational logic; hence, a part of the critical path. They determine this critical path and its effect on the clock frequency. The second approach compares their approach for a particular function of a JPEG decode algorithm with a data cache with estimated access latencies. They show that the SMMU is not as effective as a data cache in ideal conditions, they are much better in the worst case. The third approach is a case-study implementation of the entire JPEG decode algorithm with the SMMU.

- What other work is listed as future work? Integrating one of the SPM allocation techniques with the SMMU to determine the where to place OPEN and CLOSE commands.

Additional questions:

- What are the limitations/assumptions of this work?
  The user must specify the size of the object and the physical address. They claim that there are algorithms that determine the best physical address, but I haven't read that work yet.
- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, ...)
  They require the following modifications: 1) hardware, 2) WCET analysis, and 3) compilers to generate the OPEN/CLOSE commands.

5.3 Interconnect

Real-Time Control of I/O COTS Peripherals for Embedded Systems [66, 10]

- What is the background of this work? What is the motivation?
  Due to mass production, commercial-off-the-shelf (COTS) peripherals are not only cheaper than custom made systems, but also provide performance orders of magnitude higher (e.g. PCI Express vs. the real-time SAFEbus). It is thus tempting to employ COTS components in real-time systems. However, their unpredictable timing makes their use in such systems difficult: they are typically designed paying little attention to worst-case timing behavior. This paper considers the I/O subsystem. Modern real-time systems may include several high-bandwidth I/O devices. Connecting these devices to the processor through COTS interconnects yields unpredictable delays and may cause deadline misses. In contrast to CPus, where real-time scheduling is common, it is not supported by COTS interconnect systems such as the PCI bus.
- What is the main goal?
  The goal is to enable the use of COTS interconnect in real-time systems.
- What did they do to achieve this goal?
  They designed a real-time I/O management system. This system consists of a real-time bridge for each COTS peripheral connecting the peripheral to the COTS interconnect, and a reservation controller. The reservation controller decides at any point in time which real-time bridge may access the COTS intereconnect. It can implement a variety of real-time scheduling policies, such as EDF or RM. They extend the Real-Time Calculus to determine I/O delay bounds and each bridge's necessary buffer size to guarantee lossless traffic delivery.
- How do they evaluate their approach? Does it achieve the goal? Do they compare it with other work?
  They evaluate their approach through measurements of synthetic benchmarks on a COTS PC platform. As the speed of the PCIe is running at 2.5 GHz is very high, they extended the reservation controller to poll the state of the bus at a resolution of one microsecond. They also reduced the FSB frequency to achieve a typical bandwidth value for embedded platforms. The synthetic benchmark consists of four real-time flows competing for main memory. The tasks' periods are harmonic, and the total utilization is 100%. So the task set is schedulable under RM. First, they execute this task set on an unmodified system. One of the tasks misses its deadline at a near-critical instant. Once the real-time I/O management is employed, no deadline misses are observed.

- What other work is listed as future work?
  In future work, they want to distinguish traffic from the same peripheral, i.e. giving different priorities to different traffic from the same peripheral. They also plan to provide hardware-based preprocessing and filtering on the real-time bridge to drop less important packets if the main CPU is overloaded.

Additional questions:

- What are the limitations/assumptions of this work?
  Requires redevelopment of drivers for peripherals on host and real-time bridge side.
- Which parts of a system/design process are modified by this work? (e.g. hardware (which feature?), WCET analysis, scheduling, compiler, programming language, . . .)
  Hardware: real-time bridges and reservation controller. Software: drivers for bridges and hosts.

5.4 Academia

5.5 Industry

Chapter 6

Conclusion and Future work

6.1 Summary of Results

This is my summary

6.2 Publications

6.3 Future Work

Here is what you can keep doing
Talk about future research challenges for a predictable architecture.

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

Bibliography

[1] GNU ARM Toolchains.

[2] 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.

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

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

[10] S. Bak, E. Betti, R. Pellizzoni, M. Caccamo, and L. Sha. Real-time control of I/O COTS peripherals for embedded systems. In *RTSS '09: Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, pages 193–203, Washington, DC, USA, 2009. IEEE Computer Society.

[11] 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.

[12] 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.

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

[14] 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.

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

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

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] S. Craven, D. Long, and J. Smith. Open source precision timed soft processor for cyber physical system applications. In *Proceedings of the 2010 International Conference on Reconfigurable Computing and FPGAs*, RECONFIG '10, pages 448–451, Washington, DC, USA, 2010. IEEE Computer Society.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

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

[46] 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.

[47] 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.

[48] 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.

[49] C.-G. Lee, J. Hahn, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium*, page 264, Washington, DC, USA, 1996. IEEE Computer Society.

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

[51] 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.

[52] 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.

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

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2/3):217–247, 2000.

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

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

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

[63] 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.

[64] 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.

[65] H. D. Patel, B. Lickly, B. Burgers, and E. A. Lee. A timing requirements-aware scratchpad memory allocation scheme for a precision timed architecture. (UCB/EECS-2008-115), Sep 2008.

[66] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of CPU and I/O transactions in COTS-based embedded systems. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 221–231, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[69] 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.

[70] 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.

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

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

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

[74] 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.

[75] 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.

[76] 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.

[77] 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.

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

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

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

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

[82] 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.

[83] B. J. Smith. *Architecture and applications of the HEP multiprocessor computer system*, pages 342–349. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[84] 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.

[85] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. Wcet centric data allocation to scratchpad memory. *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 10 pp.–232, Dec. 2005.

[86] 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.

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

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

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

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

[91] 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.

[92] 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.

[93] 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.

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

[95] J. Whitham and N. Audsley. Forming virtual traces for WCET analysis and reduction. In *RTCSA '08: Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 377–386, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

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

[99] 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.

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

[101] Xilinx. *Core generator guide*.

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

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

[104] 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.

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