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

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 Hiren Patel
I want to thank Jan Rieneke

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 as background

Fig. 1.1 shows an image

1.2 Intro Section Header 2

Here is another header

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 6, Chapter 7,

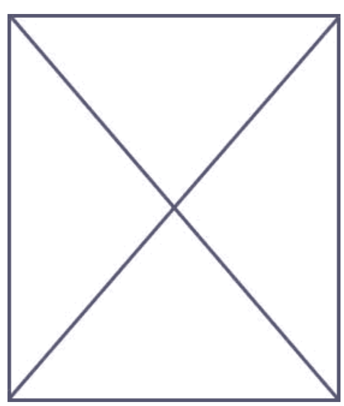


Figure 1.1: Image Placeholder

Chapter 2

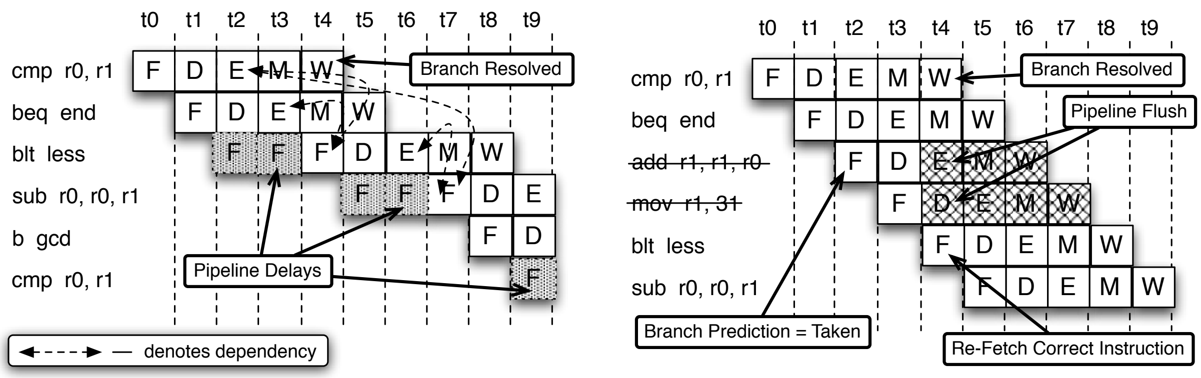
Precision Timed Machine

In this chapter we present the PREcision Timed (PRET) Machine. It is important to understand why and how current architectures fall short of timing predictability and repeatability. Thus, before we preset the PRET architecture, we briefly 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. The introduction of pipelining vastly improved architecture average-case performance. It allowed faster clock speeds for processors, and improved the instruction throughput compared to single cycle architectures. Pipelining the datapath allowed subsequent instructions to begin execution while prior instructions were still being completed. 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. The handling of these hazards an important factor to the timing predictability and repeatability of the architecture design. To illustrate this point, we discuss basic hardware additions proposed to reduce performance penalty from hazards, and their effects on execution time and predictability.

```
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 2.2: Sample code for GCD with conditional branches

We began by looking at how control-flow changes are handled in pipelines. 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 adds more complexity(Todo: ?), as whether or not the branch is taken depends on an additional condition bit. The code segment in figure 2.2 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 instructions in ARM branch based on conditional flags that are set with special compare instructions(Todo: citation). The *cmp* instruction is one



GCD without branch prediction

GCD with branch prediction (mispredict)

Figure 2.1: Handling of conditional branches in single threaded pipelines

such compare instruction that subtracts two registers and updates the conditional flags according to the results. The GCD implementation shown in the code uses this mechanism to determine whether to continue or end the algorithm. Figure 2.1 show two ways branches are commonly 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 the pipeline – fetch (F), decode (D), execute (E), memory (M) and writeback (W).

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 2.1. 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 architecture used contain 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 by design.

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 2.1 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. In the case of the ARM ISA, the analysis simply accounts for the branch penalty after every 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.

Data hazards occur when instructions depend on the results of previous instructions that have yet to commit. The code segment shown in figure 2.3 contains instructions that each depend on the result of its previous instruction. The top of figure 2.4 shows an ex-

```
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 2.3: Sample code with data dependencies

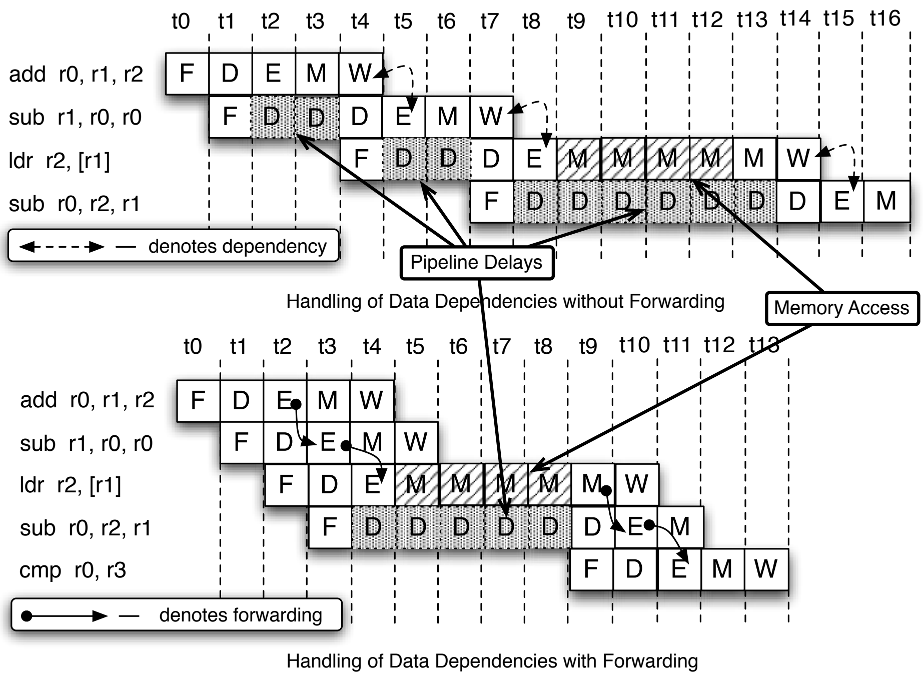


Figure 2.4: Handling of data dependencies in single threaded pipelines

ecution of the code segment on a naive pipeline without handling of data hazards. Data hazards in this case are handled by inserting pipeline delays to ensure the completion of all dependent instructions. Similar to inserting pipeline delays of control-flow hazards, this method allows for predictable static execution time analysis, but at a slight cost of performance. Pipeline forwarding is the most common way of handling data hazards that occur from pipelining. A pipeline forwarding circuitry consists of backwards paths for data from different pipeline stages to the inputs of arithmetic units, and multiplexers to select amongst them. It provides a way to directly access computation results from the previous instruction before it commits. 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 2.4 shows the execution with forwarding in the pipeline. No pipeline bubbles are needed for the first *sub* instruction and *ld* instruction, as the results they depend on can be computed in one cycle by the ALU, and forwarded through the forwarding paths. Notice that although there is dynamic execution in the pipeline forwarding circuitry, it is actually possible to statically predict the execution time accurately. The logic in the pipeline controller that enables and selects the correct forwarding bits only needs to check a small set of previous instructions to detect data-dependencies. 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) The internal state of the branch predictor on the other hand is dependent on branch histories which may have happened arbitrarily long ago in the execution sequence, (Todo: more complicated ways of handling data hazards include out-of-order execution…)

We assume the *ld* instruction accesses results in a cache miss, requiring access to main

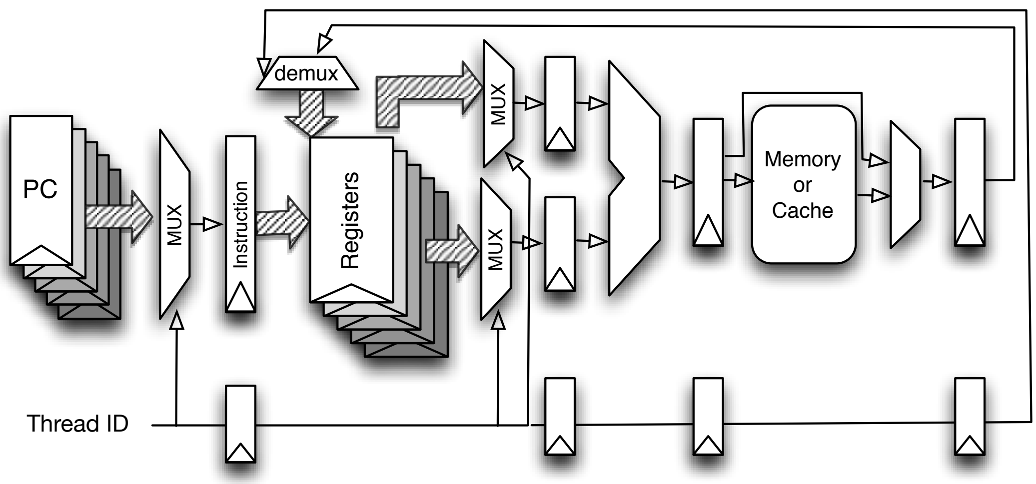


Figure 2.5: Simple Multithreaded Pipeline

memory. Stalls are still inserted for the the second *sub* instruction, as it is waiting upon the results of a memory operation. 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. Typical memory access latencies could range from 10 to 1000 cycles. We can thus see that forwarding 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.

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 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. [22] 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.

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 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, and barely any hardware is required to implement round robin thread scheduling; a simple $log(n)$ bit

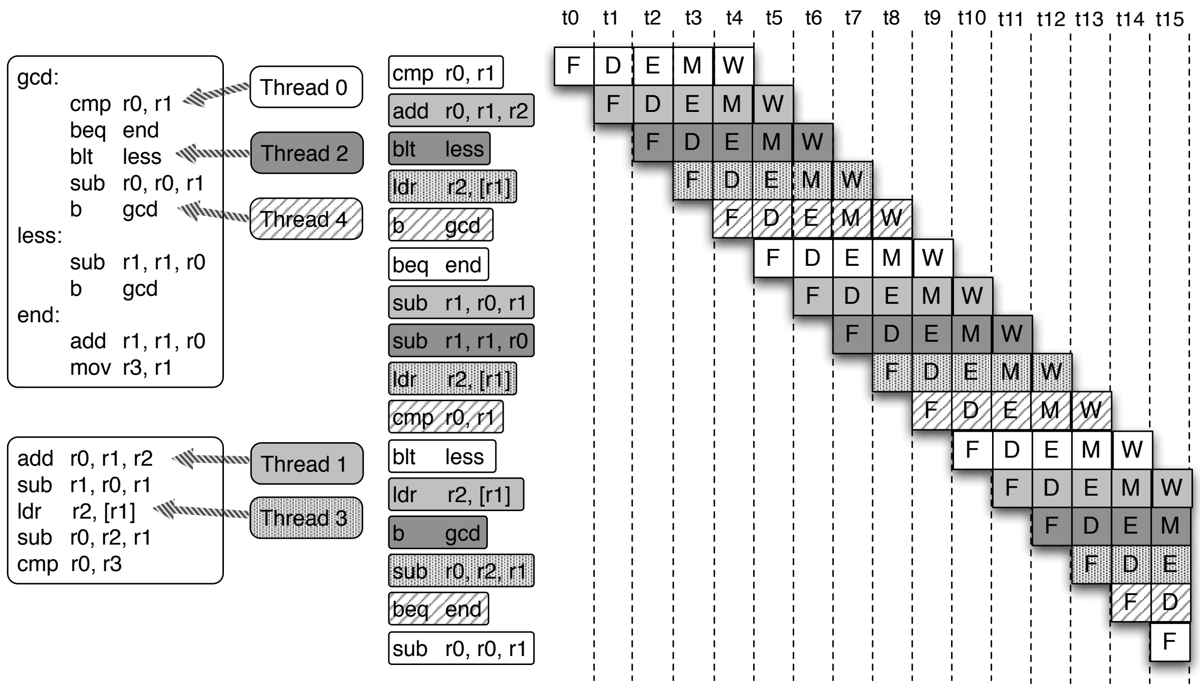


Figure 2.6: Sample execution sequence of a thread-interleaved pipeline with 5 threads and 5 pipeline stages

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 2.2 and figure 2.3 are being executed in this pipeline. Threads 0, 2 and 4 execute GCD (figure 2.2) and threads 1 and 3 execute the data dependent code segment (figure 2.3). 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 unique property of thread-interleaved pipelines, which provides the basis for a timing predictable architecture design.

For thread-interleaved pipelines, if there are, at a minimum, 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 2.1 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 2.3 is executed on thread 1 of the thread interleave pipeline in figure 2.6. The pipeline stalls inserted from top of figure 2.4 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 2.4. 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. This would however require non-trivial hardware additions to support and manage multiple write-back paths in the pipeline so contention can be avoided with the existing executing threads. Thus, in our design we simply replay the instruction for write-backs to simplify design and piggy back on the existing write-back datapath. We showed a memory access instruction in our example, but the same reasoning is applied to floating point instructions or any long latency instruction. Conventional thread-interleaved pipelines typically mark threads inactive when they are waiting for long latency operations. Inactive threads are not fetched and executed in 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.

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

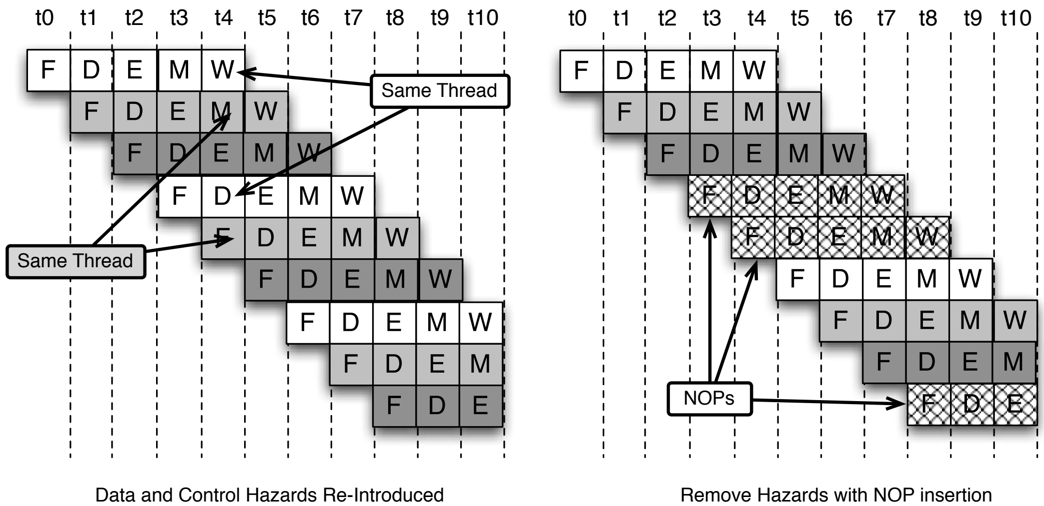


Figure 2.7: Execution of 5 threads thread-interleaved pipeline when 2 threads are inactive

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. Any analysis needs to model and explore all possible combinations of program state of all threads, which is typically infeasible. The same difficulty is present for timing analysis on most coarse-grained multithreaded architecture. It is important to understand that it is not the thread scheduling that is non-predictable, but how thread switches are triggered. It is still possible for a predictable architecture to support the scheduling of hardware threads, but it must be done statically in software, instead of dynamically in hardware. If the hardware can provide a means of explicitly setting thread schedules in software, then it is possible to make the thread schedules transparent to any timing analysis. The software still needs to be carefully constructed, since it is still possible for the software to switch hardware thread schedules dynamically, complicating the timing analysis. One could envision a program that contains different program states, each state executing a different number of threads. As long as the state transitions are statically analyzable, the timing analysis complexity is not increased. Thus, our thread-interleaved pipeline does not mark threads inactive on long latency operations, but simply replays the instruction whenever the thread is fetched, until the long latency operation completes. 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 some of the benefits of latency hiding, as other threads are still executing during the long latency operation.

Thread-interleaved pipelines however still contain structural hazards, hazards that occur when hardware units are needed by two or more instructions at the same time. Although each pipeline stage is occupied by different hardware threads, two or more threads can simultaneously be issuing *ld* instructions to the main memory. If the shared main memory can only handle one request at a time, then the second request must be queued waiting for the first requester to complete.

This creates timing interference between the threads, because the memory access time of a memory request from a particular thread now depends on if the memory is currently being access by other threads, and how many threads are already queued up. We will discuss how our redesigned memory controller supports pipelining memory access in section 2.2, but the same hazard applies to any shared hardware unit such as floating point units etc. If the hardware unit can be pipelined and accept inputs every cycle, then no contention arises between the hardware threads, and no structural hazards occur. If pipelining cannot be achieved, then any timing analysis of that instruction must include a conservative estimation that accounts for thread access interference. A time division multiplex access (TDMA) schedule can be enforced to decouple the access time of threads to the shared hardware unit.

In summary, blah blah blah...

2.2 Memory System

Chapter 3

Programming Models

Intro text here

3.1 PRET Programming model Section Header

Fig. 3.1 shows an image

3.2 Pret Programming model Section Header 2

Here is another header

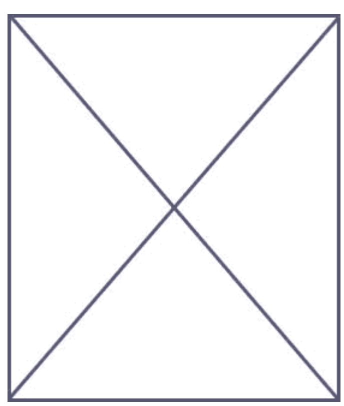


Figure 3.1: Image Placeholder

Chapter 4

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 build a single cycle hardware shifter along with an 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 base address. 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 read 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 registers (R0 to R15) to be accessed with its instructions, with register 15 being the program counter. Any data-operation done on register 15 triggers a branch.

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 includes hardware support for several floating point operations, and implements the timing instructions introduced in chapter 3. We will first discuss the architectural details of PTARM, then present the C++ software simulator of PTARM.

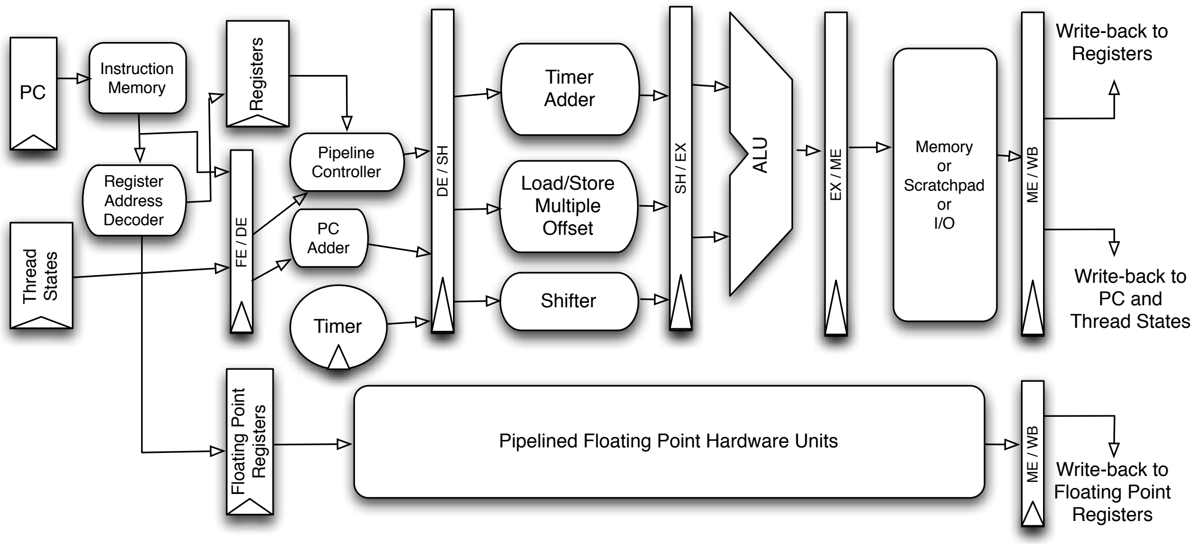


Figure 4.1: Block Level View of the PTARM pipeline

4.1 PTARM Architecture

The pipeline of PTARM implements a thread-interleaved pipeline for the ARM instruction set. PTARM is written in VHDL and targets Xilinx Virtex-5 Family FPGAs, thus a the design decisions made tailored the PTARM architecture towards optimizing the design for the Xilinx V5 FPGA. It has a 32 bit datapath and consists of a six stage pipeline and can support a minimum of six threads interleaving through the pipeline. The pipeline can be clocked up to $180MHz$ on a Virtex-5 lx110t FPGA. Figure 4.1 shows a block diagram view of the pipeline. The 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 multiple copies of the Program Counter(PC), Thread States, and Register File, which are not shown in the figure. Most of the pipeline design follows a typical Hennesy and Patterson(Todo: citation) 5 stage pipeline, but the thread interleaved pipeline also allows us to strip away the branch predictor and the data forwarding logic used to handle data-hazards. The six stages in the pipeline are – Fetch, Decode, Shift, Execute, Memory, Writeback. We will briefly describe the functional blocks of each stage, and later show how most instruction types are implemented in the pipeline.

The fetch stage of the pipeline selects the correct PC according to which thread is executing, and passes the address to instruction scratchpad. A simple $Log(n)$ bit upcounter is used to keep track of which thread current to fetch. At this point we assume the source code fits entirely on the instruction scratchpad. Later when we present the memory hierarchy of PTARM we will discuss further about the implications of larger code bases. Once the instruction is received from the scratchpad, it goes through a register address decoder to quickly determine what bits to send to the register. Typical RISC instruction sets such as MIPS have encoding of instruction bits where the register operands have a fixed location for all instruction types. Thus, once the instruction is received from instruction memory, the selected bits can be propagated as addresses to the register file. In the ARM instruction set however, not all instruction encoding have the register addresses at the same location. Thus, we insert in a simple and small logic block for a quick decoding of register addresses.

The decode stage of the pipeline consists of the pipeline controller which does the full decoding of instructions and sets the correct pipeline signals to be propagated down the pipeline. Typically the controller needs to know the current instructions in the pipeline to detect the possibility of pipeline hazards. However,in a thread-interleaved pipeline, other instructions in the pipeline belong to other 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. These signals get propagated down the pipeline stages, and they control the execution units to ensure the correct actions are taken, and signal the multiplexers to select the correct data operands. Because most of ARM instructions are conditionally executed, the pipeline controller also checks the condition bis to determine whether the instruction is to be executed or not. The PC Adder is almost just a simple adder that increments the PC. The only addition is instead only outputting an address incremented by four, it outputs two addresses, the PC incremented by both four and eight. The address calculation of ARM branch instructions add the offset to PC+8, instead of just the current PC, so we need to store the additional PC offset of eight in case of branch instructions. The Timer logic block is a hardware counter clocked to the processor clock which is used to implement the timing instructions mentioned in chapter 3. The Timer counts time in nanoseconds, and it's hardware logic does not reside in the decode stage. The time value however is latched in the decode stage as the subsequent stages use it for timer manipulation, so in figure 4.1 we include it in the decode stage. We will discuss in more detail how the timing instructions are implemented later in this chapter.

The shift stage of the pipeline is the additional stage on top of a conventional five stage pipeline. The ARM ISA data-processing instructions include shifter bits and even a shifter operand for data-processing register shift instructions to shift the operands before the logical or arithmetic operations. Thus, an extra 32 bit shifter is included to shift the operands before the execution stage. 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. We will later describe in detail how the load/store multiple instructions are executed within the pipeline. The timer adder logic block is a 32 bit add/subtract unit. Since time is represented in 64 bit values, we add an additional 32 bit add/subtract-er so the execution stage doesn't need to incorporate a 64 bit ALU.

The execute stage simply just contains the 32 bit ALU that can do both logical and arithmetic operations. The memory stage interacts with the data scratchpad and memory controller, and the write back stage writes the correct data back to the registers and hardware thread states.

The floating point hardware units are pipelined floating point computation units generated with the Xilinx Coregen tool.

4.2 Instruction Implementations

In this section we will go into more details about how each instruction type is implemented and how the hardware blocks shown in the previous sections are used. We will go through

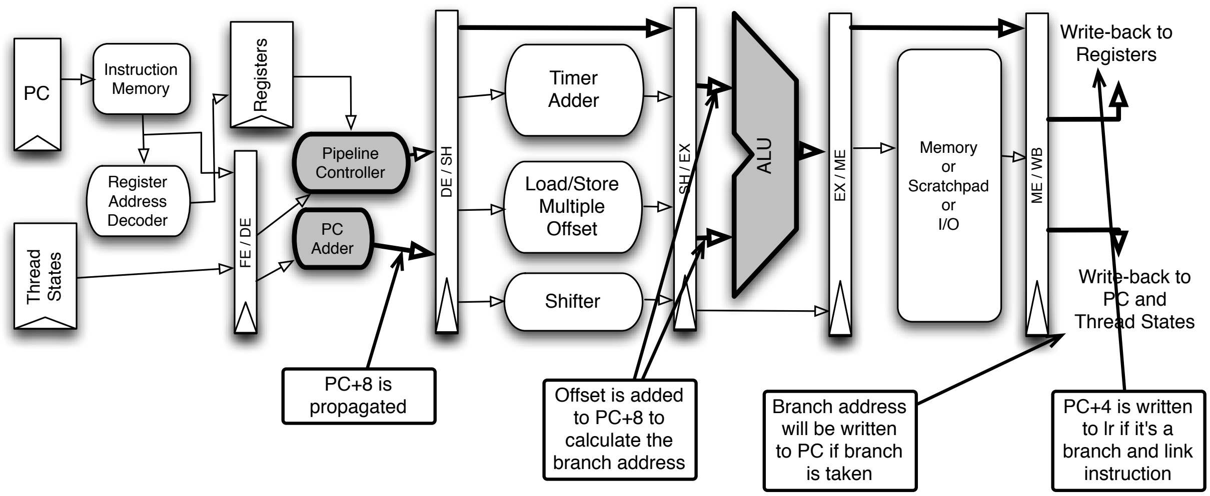


Figure 4.2: Branch Instruction Execution in the Ptarm Pipeline

different instruction types and also discuss the timing implications of our implementation. We will summarize with a table with all instructions and the cycle count it takes to execute them.

4.2.1 Branch

Branch instructions in the ARM can conditionally branch forward or backwards by up to 32MB. Because the general purpose register 15 is the PC, any write to R15 is also considered a branch instruction that branches to the address written into R15. Figure 4.2 show how the branch instruction is executed in the thread-interleaved pipeline. The hardware units involved are highlighted in gray and the lines are in bold. The branch instructions for the ARM ISA calculate the address based upon an offset added to PC incremented by 8. Thus, the PC adder, in addition to incrementing the PC by 4, also increments the PC by 8 to pass to the ALU for correct address calculation. Once the address is calculated, it is written back to the PC for the corresponding thread. If the instruction is a branch and link, PC+4 will also be written back to the link register (R14). In the case of a conditional branch, the PC is only written back if the condition holds true, or else PC+4 is written back to the PC. Because of the thread interleaved pipeline, we are not concerned that the pipeline will be stalled waiting for result of the branch to execute the next instruction. Instead, instructions from other threads will be fetched before the results of the branch is needed.

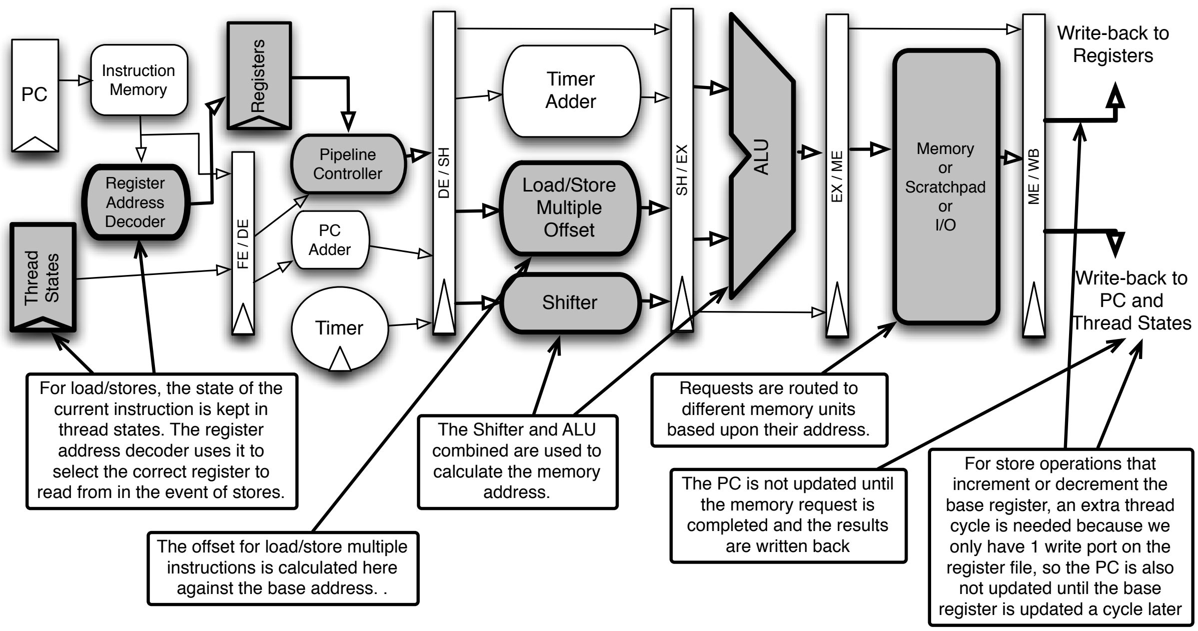


Figure 4.3: Load/Store Instruction Execution in the Ptarm Pipeline

4.2.2 Data-Processing

4.2.3 Floating Point

4.2.4 Timing Instructions

4.2.5 Loads and Stores

4.3 PTARM Simulator

4.4 Worst Case Execution Time Analysis

Chapter 5

Related Work

5.1 Architectural Modifications

Fig. 5.1 shows an image

5.2 Related Section Header 2

Here is another header

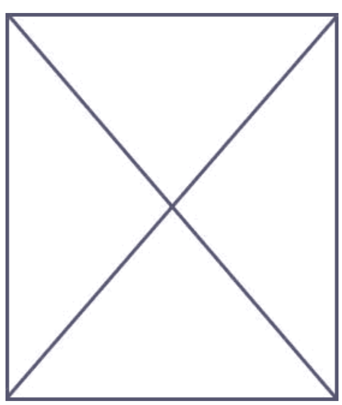


Figure 5.1: Image Placeholder

Chapter 6

Applications

6.1 Eliminating 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 [14] 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 [14]. 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 [6, 9, 25] 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. For example, 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 [3, 20, 2, 8].

In order to not be confused with Kocher's [14] 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 [17, 13], fault injection attacks [4, 10], and many others [25].

In recent years, we have seen a tremendous effort to discover and counteract side-channel attacks on encryption systems [4, 8, 15, 11, 1, 12, 7, 24, 23]. However, it is difficult to be fully assured that all possible vulnerabilities have been discovered. The plethora of research on side-channel exploits [8, 4, 15, 11, 1, 12, 7, 24, 23] 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. [8] 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 [14] or blinding signatures [14, 7]. Other techniques such as branch equalization [18, 25] 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.

In this paper, we present the PREcision Timed (PRET) architecture [16] in the context of embedded cryptosystems, and show that an architecture designed for predictability and controllability effectively eliminates all time-exploiting attacks. Originally proposed by Lickly et al [16], PRET provides instruction-set architecture (ISA) extensions that allow programmers to control an algorithm's temporal properties at the software level. To guarantee that the timing specifications are honored, PRET provides a predictable architecture that replaces complex pipelines and speculation units with multithread-interleaved pipelines, and replaces caches with software-managed fast access memories. This allows PRET to maintain predictability without sacrificing performance. We target embedded applications such as smartcard readers [15], key-card gates [5], set-top boxes [15], and thumbpods [21], which are a good fit for PRET's embedded nature. We demonstrate the effectiveness of our approach by running both the RSA and DSA [19] 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.

6.2 Real Time 1D Computational Fluid Dynamics Simulator

Here is another header

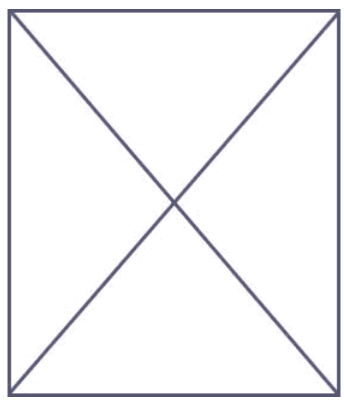


Figure 6.1: Image Placeholder

Chapter 7

Conclusion and Future work

7.1 Summary of Results

This is my summary

7.2 Future Work

Here is what you can keep doing

Bibliography

[1] 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.

[2] 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.

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

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

[5] 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.

[6] 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.

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] 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.

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

[23] 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.

[24] 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.

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