A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance

Isaac Liu[1] Jan Reineke[2] David Broman[1,3] Michael Zimmer[1] Edward A. Lee[1]

liuisaac@eecs.berkeley.edu, reineke@cs.uni-saarland.de, {broman,mzimmer,eal}@eecs.berkeley.edu

[1]University of California, Berkeley, CA, USA [2]Saarland University, Germany [3]Linköping University, Sweden

***Abstract*—We contend that repeatability of execution times is crucial to the validity of testing of real-time systems. However, computer architecture designs fail to deliver repeatable timing, a consequence of aggressive techniques that improve average-case performance. This paper introduces the Precision-Timed ARM (PTARM), a precision-timed (PRET) microarchitecture implementation that exhibits repeatable execution times without sacrificing performance. The PTARM employs a repeatable thread-interleaved pipeline with an exposed memory hierarchy, including a repeatable DRAM controller. Our benchmarks show an improved throughput compared to a single-threaded in-order five-stage pipeline, given sufficient parallelism in the software.**

I. Introduction

Can we trust that a processor repeatedly performs correct computations? The answer to this question depends on what we mean by *correct computations*. In the core abstraction of computation, rooted in von Neumann, Turing, and Church, correctness refers only to correct transformation of data. Execution time is irrelevant to correctness; it is instead a performance factor, a quality metric where faster is better, not more correct. However, in cyber-physical systems, which combine computation (embedded systems), networks, and physical processes, execution time is often a correctness criterion [1].

Although extensive research has been invested in techniques for formal verification, testing is in practice the dominating method to gain confidence that a system is working correctly according to some specification. However, without repeatability, testing proves little. Although vital in embedded and real-time systems, repeatable timing is also valuable in general-purpose systems. For concurrent programs, in particular, non-repeatability is an obstacle to reliability [2].

ACCEPTED VERSION. In Proceedings of the 30th IEEE International Conference on Computer Design (ICCD 2012), Montral, Quebec, Canada, 2012, IEEE. Published version: http://dx.doi.org/10.1109/ICCD.2012.6378622

© 2012 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), the U. S. Army Research Laboratory (ARL #W911NF-11-2-0038), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota. The third author was funded by the Swedish Research Council #623-2011-955.

Repeatable timing is easy to achieve if you are prepared to sacrifice performance. The engineering challenge is to enable both timing repeatability *and* performance. Conventional architectures with caches and pipelines improve average-case performance, but make execution time inherently non-repeatable.

In previous work [3], [4] we outline ideas for achieving repeatable timing by using a thread-interleaved pipeline, replacing caches with programmable scratchpad memories, and designing a DRAM controller with predictable timing. In this paper we present and evaluate a concrete implementation of this approach for achieving repeatable timing with competitive performance.

Repeatability of timing can be viewed at different levels of granularity. At a coarse-grained level, repeatability means that for a given input, a program always yields the same execution time. At the most fine-grained level, each processor instruction always takes the same amount of time. Excessively fine-grained constraints on the execution time lead to design decisions that sacrifice performance. On the other hand, an excessively coarse-grained solution may make program fragments non-repeatable due to context dependencies. In this work we confront this tradeoff, and show that a microarchitecture with repeatable timing and competitive performance is feasible. This enables designs that are assured of the same temporal behavior in the field as exhibited on the test bench. To be specific, in this paper we make the following contributions:

- To enable repeatability and avoid pipeline hazards, we design and implement a *thread-interleaved pipeline* for a subset of the ARMv4 ISA. The architecture is realized as a soft core on a Xilinx Virtex 5 FPGA (Section III-A).
- We extend previous work on predictable DRAM controllers [5] to make DRAM memory accesses repeatable. We discuss the tradeoff between fine-grained timing repeatability and average-case performance (Section III-C).
- We evaluate the performance of the architecture by comparing with both a conventional ARMv2a and an ARMv4 architecture. With the assumption of parallelizable tasks or independent tasks, we show that the thread-interleaved pipeline achieves significant performance gains (Section IV).

II. Related Work

There is a considerable body of recent work [6], [7], [8] on the related problem of building *timing predictable* computer architectures. Timing predictability is concerned with the

ability to statically compute safe and precise upper bounds on execution times of programs. In contrast, timing repeatability is concerned with the ability to repeat timing. A system may be predictable, yet non-repeatable, e.g., due to pipelining or caching. Similarly, a system's timing may be repeatable, yet hard to predict statically. The PTARM architecture presented in this paper is both timing repeatable as well as timing predictable.

Several architectures have been proposed that exhibit repeatable timing. Whitham [9] and Schoeberl [10] both use microcode implementations of architectures to achieve repeatable timing. Whitham introduces the Microprogrammed Coarse Grained Reconfigurable Processor (MCGREP) [9], which is a reconfigurable predictable architecture. MCGREP uses microcode to implement pipeline operations on a simple two stage pipeline with multiple execution units. The pipeline stores no internal state and no caches are used, so each microcode operation takes a fixed number of cycles, unaffected by execution history.

Schoeberl presents the Java Optimized Processor (JOP) [10]. JOP uses a two-level stack cache architecture to implement the stack based architecture of JavaVM. It implements a three-stage pipeline and uses two registers to store the top two entries of the stack; the remaining stack is stored in the SRAM. No branch predictor is used, as only a small branch delay penalty is incurred. All bytecode on JOP is translated into fixed-length microcode, and each microcode executes in a fixed number of cycles, independent of its surrounding instruction.

Andalam *et al.* [11] propose the Auckland Reactive PRET (ARPRET), designed to execute compiled PRET-C programs. PRET-C is a C extension (via macros), with certain restrictions, supporting synchronous concurrency and high-level constructs for expressing logical time. ARPRET consists of a customized three-stage Microblaze [12] soft processor core and a Predictable Functional Unit (PFU) for hardware scheduling. ARPRET uses only on-chip memory, so read/write memory instructions take one clock cycle. ARPRET has currently no support for memory hierarchies or multiple cores.

Our proposed architecture uses thread-interleaved pipelines, which have been proposed and employed in various architectures by research and industry. The CDC6600 [13], Lee and Messerschmitt [14], the Denelcore HEP [15], the XMOS XS1 architecture [16], the Parallax Propeller Chip [17] and the Sandbridge Sandblaster [18] all use fine-grained thread interleaving for different applications. Lee and Messerschmitt [14] use a round-robin thread scheduling policy while the Sandblaster [18] uses a token-triggered threading policy. The XMOS XS1 architecture [16] allows hardware threads to be dynamically added and removed from the thread scheduling, however, this causes each thread's execution frequency to vary depending on the number of threads executing at one time. Unlike previous approaches, our thread-interleaved pipeline uses a static round-robin thread scheduling policy with memory hierarchy support to achieve repeatable performance.

III. Precision-Timed ARM

PTARM is a concrete implementation of a precision-timed (PRET) machine [4], a computer architecture designed for predictable and repeatable performance. PTARM implements a subset of the ARMv4 ISA [19] and does not support thumb mode, an extension that compacts the instructions to 16 bits, instead of the typical 32 bits. Conventional architectures use complex pipelines and speculation techniques to improve performance. This leads to non-repeatable behaviors because the instruction execution is affected by the implicit state of the hardware. PTARM improves performance through predictable and repeatable hardware techniques. These include a refined thread-interleaved pipeline, an exposed memory hierarchy, and a repeatable DRAM memory controller. In this section we give an overview of the PTARM architecture, and discuss how repeatability is achieved.

A. A Thread-Interleaved Pipeline

Thread-interleaved pipelines fully exploit *thread-level parallelism* (TLP) by using a fine-grained thread switching policy; every cycle a different hardware thread is fetched for execution. PTARM implements an in-order, single-issue five-stage pipeline, similar to a conventional five-stage RISC pipeline. State is maintained for each thread in the pipeline to implement multithreading. A round-robin thread scheduling policy is used to reduce the context-switch overhead to zero and to maintain repeatable timing for all hardware threads.

By interleaving enough threads, *explicit dependencies* between instructions within the pipeline can be completely removed in thread-interleaved pipelines. Explicit dependencies are dependencies that arise from the flow of data at the instruction level, such as data dependencies that depend on the register values, or control dependencies that depend on a branch address. In general, by interleaving the same number of threads as pipeline stages in a round-robin fashion, the explicit dependencies are removed because each instruction in the pipeline belongs to a different hardware thread. PTARM interleaves four threads in a five-stage pipeline, similar to Lee and Messerschmitt [14], by writing back the next program counter (PC) before the writeback stage; this ensures that the next instruction fetch from the same thread always fetches the correct address without needing to stall.

Control hazards are completely removed in the pipeline in this way, as branch instructions are always completed before the next instruction from the same hardware thread is fetched. When an exception occurs, no instruction needs to be flushed from the pipeline, because exceptions are thread specific; the other instructions in the pipeline belong to other threads. No instruction is speculatively executed in PTARM. The removal of explicit dependencies within the pipeline also allows us to have a simpler pipeline design and to strip out the branch predictor and data forwarding logic. This can improve the clock frequency of the pipeline [20].

Long latency operations, such as memory operations, can still create data hazards within the pipeline. Conventional multithreaded architectures disable the scheduling of hardware threads that are waiting for memory. However, this leads to

non-repeatable timing, because the execution frequency of the hardware threads can change depending on the execution context of other hardware threads. Furthermore, the number of active threads can drop below the minimum requirement to remove hazards. Thus, PTARM does not dynamically deactivate threads when waiting for memory, but simply replays the instruction until it is complete. Although this slightly reduces the throughput of the pipeline, the latency hiding benefits of multithreading are still present, and repeatable timing is achieved.

The static round-robin schedule provides a consistent latency between instruction fetches from the same thread, since a constant number of instructions will be fetched in between. The term *thread cycle* is used to encapsulate this latency, and simplify the numbers for timing analysis. Intuitively, a thread cycle abstracts away the processor cycles in between instruction fetches from the perspective of the hardware thread. For PTARM, each thread cycle is equivalent to four processor cycles, as four hardware threads are interleaved through the pipeline.

The hardware threads on a multithreaded architecture, by definition, share the underlying pipeline datapath and any hardware unit implemented in it. The sharing of hardware units can create *implicit dependencies* between different hardware threads. One example of this is a shared branch predictor; the execution time of branch instructions can implicitly depend on previous branch instructions executed. This effect can also be observed across hardware threads if the threads share the same branch predictor, which leads to non-repeatable timing on multithreaded pipelines. In PTARM, the branch predictor is not needed, but more importantly, no shared hardware unit contains state information that can affect the execution time of other instructions. We will discuss how this is done for the memory system in the next two sections.

B. Scratchpad Memory

Conventional memory hierarchies use caches to bridge the latency gap between main memory and the pipeline. However, caches hide the memory hierarchy from the programmer by managing their contents in hardware. This leads to non-repeatable timing, because the execution time of memory operations depends on the state of the cache, which cannot be explicitly controlled by the the programmer.

PTARM *exposes* the memory hierarchy to the programmer by using scratchpads [21] that map to distinct regions of memory. Scratchpads use the same memory technology (SRAM) as caches, but do not implement the hardware controller to manage their memory contents. The programmer or compiler explicitly manages the contents on the scratchpad in software through static or dynamic scratchpad allocation schemes. By exposing the memory hierarchy to the programmer, the memory access latency for each memory request depends only on the accessed address, and not on the state of a hardware controller. Thus, repeatable memory access latencies are achieved.

C. Dynamic RAM

Conventional memory controllers *do not* provide repeatable latencies for DRAM accesses for two reasons:

1) The latency of a memory access depends on the access history to the memory, which determines whether or not a different row has to be activated. If several tasks share the memory, this access history is the result of the interleaving of the access histories of the different tasks. From the perspective of an individual task, the access history, and thus the access latencies depend on the behavior of other tasks, and are thus non-repeatable.
2) DRAM cells have to be refreshed periodically. Conventional memory controllers may issue refreshes and block current requests at—from the perspective of a task—unpredictable and unrepeatable times.

In previous work [5], we have introduced a PRET DRAM controller, which provides *predictable* access latencies and *temporal isolation* between different clients. The focus of this section is on how to integrate the PRET DRAM controller within PTARM to also provide *repeatable* access latencies.

To do so, we need to briefly recapitulate the design of the controller from [5].

1) PRET DRAM Controller: Conventional DRAM controllers abstract the DRAM device they provide access to as a single resource. In contrast, the PRET DRAM controller views a DRAM device not as a single resource to be shared entirely among a set of clients, but as several resources which may be shared among one or more clients individually. In particular, we partition the physical address space following the internal structure of the DRAM device, i.e., its ranks and banks. Similar to the thread-interleaved pipeline, the memory controller pipelines accesses to the blocks of this partition in a time-triggered fashion, thereby alternating between the two ranks of the DRAM device. This eliminates contention for shared resources within the device, making accesses temporally isolated. A closed-page policy eliminates the influence of the access history on access latencies.

2) Integration within the PTARM Microarchitecture: The four resources provided by the backend of the DRAM controller are a perfect match for the four hardware threads in PTARM's thread-interleaved pipeline. We assign exclusive access to one of the four resources to each hardware 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.

During the time of a DMA transfer, the initiating thread can continue processing and accessing the instruction and data scratchpads. 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. 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].

Without further adaptation, latencies of DMA transfers and loads still exhibit slight variations, resulting in non-repeatability, for the following two reasons:

1) Both the thread-interleaved pipeline and the pipelined access scheme to the DRAM operate periodically, however, on different periods. As a consequence the latency for individual loads may vary between 3 and 4 thread cycles, depending on the alignment of the two periodic schemes.
2) Accesses may interfere with refreshes, which need to be issued to every row of the DRAM at least every 64 ms. The interference by refreshes can be partly eliminated, as discussed in previous work [5].

We solve this problem, simply by slowing down every load and every DMA transfer to its worst-case latency. As a consequence, every load will take 4 thread cycles, and every DMA transfer will experience the worst-case latency determined in our previous work [5]. As the possible variation in latencies is small, at most one thread cycle, the impact of this measure is quite small as well.

Stores are fundamentally different from loads in that a hardware thread would not have to wait until the store has been performed in memory. In PTARM, we add a single-place store buffer to the frontend. This store buffer can usually hide the store latency from the pipeline: Specifically, using the store buffer, stores which are not succeeded by other loads or stores can be performed in a single thread cycle. Other stores 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. Note that while a store buffer introduces variable latency stores, it does not break repeatability at the task level: on the same inputs, a task will generate the same sequence of loads and stores, which will in turn experience the same repeatable latency. The only requirement for this is that the instruction immediately preceding the task's execution is not a store, which is usually the case, and, if not, can easily be enforced. On the other hand, to achieve repeatability at the level of individual memory instructions we would have to slow down *every* store to two thread cycles. We believe that the associated gain in fine-grained repeatability (instruction level) does not justify the significant loss in performance.

D. Repeatable Timing

With the refined thread-interleaved pipeline, exposed memory hierarchy, and adjusted DRAM controller—PTARM achieves fine-grained *repeatable timing*. That is, given the same inputs to a specific program fragment (for example a task in a real-time system), the same timing behavior will be observed for each run, thus giving a deterministic execution time of the task. *Predictability*, by contrast, concerns the ability to compute (statically) a safe and tight upper bound of the worst case execution time. For example, a code fragment that executes in 200 to 300 cycles for the same data input may be predictable (if we can compute an upper bound of WCET), but is not repeatable (execution time varies).

The meaning of the word *input* is vital for the definition of repeatable timing. We define inputs as the values (for example in memory or registers) that can be explicitly controlled by software in the program. Architecture states, such as the memory controller or pipeline state, are typically not explicitly controllable in software, and thus do not fit in our definition of inputs[2].

PTARM is repeatable at a fine-grained level because each of its instructions exhibit timing behaviors independent of architecture state. Table I shows the instruction execution times in *thread cycles*, assuming the DRAM controller operates at double the frequency of the pipeline.

TABLE I
LATENCIES OF SELECTED PTARM INSTRUCTIONS (IN THREAD CYCLES).

Instructions	**Latency** (thread cycles)	
Data Processing	1	
Branch	1	
DMA operations	1	
	SPM	DRAM
Load Register (*offset*)	1	4
Load Register (*pre/post-indexed*)	2	5
Store Register (*all*)	1	$1\text{-}2^{\delta}$
Load Multiple (*offset*)	N_{reg}	$N_{reg} \times 4$
Load Multiple (*pre/post-indexed*)	$N_{reg}+1$	$(N_{reg} \times 4)+1$
Store Multiple (*all*)	N_{reg}	$N_{reg} \times 2$

N_{reg}: This is number of registers in the register list.
$^{\delta}$: The store buffer can hide the store latency to DRAM, reducing it to a single thread cycle. In case of stores succeeded by other loads or stores, however, the store latency increases to two thread cycles.

All data processing and branch instructions take only a single thread cycle. The execution time of instructions after a branch are not affected by the control flow change, because the branch will already have been committed. With an exposed memory hierarchy, the memory instruction latencies depend only on the region of the access, and not the state of a cache or hardware controller. This is reflected in Table I, as memory instruction access to DRAM or SPM will exhibit different timing properties. For a specific program input, the memory addresses accessed will be the same for a program, preserving the repeatable timing property. Load/store multiple instructions in ARM issue multiple memory operations on registers in a single instruction. Although load/store multiple instructions have variable latencies, the latency of each instruction depends only on the number of registers operated on and the memory region accessed. Because the list of registers operated on is statically encoded as part of the instruction, repeatable timing is also preserved for such programs.

IV. EXPERIMENT AND RESULTS

To compare resource utilization and performance of the PTARM architecture to other architectures, we synthesize a preliminary implementation of PTARM as a soft core onto a Xilinx Virtex-5 XC5VLX110T FPGA [22]. The benchmarks

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

[2]If architecture states are, however, controllable in software, a processor with a cache can become repeatable at a coarse-grained level. For example, if the cache is always explicitly flushed before executing a task, the execution time for the task may be repeatable.

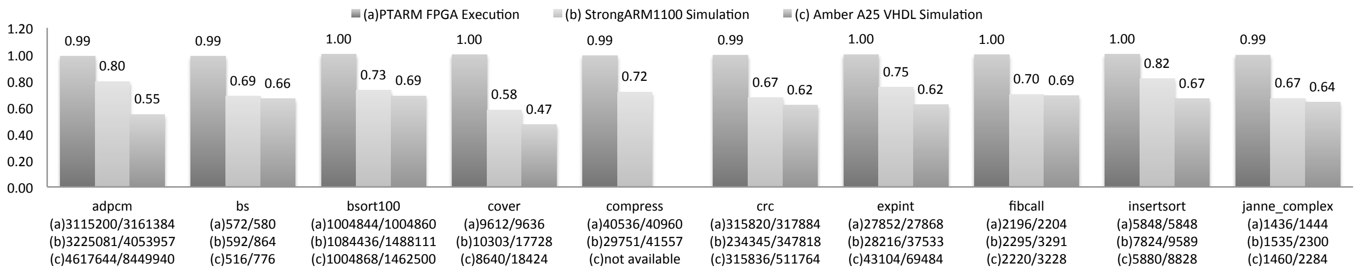

Fig. 1. Instruction throughput (instructions/thread cycles) of Mälardalen WCET benchmarks.

chosen for the performance comparison fit entirely within the scratchpad of PTARM. This is intentional, as a full-system evaluation would be affected by several factors beyond the scope of this paper, such as optimized scratchpad allocation schemes.

A. Resource Utilization

We compare resource utilization of PTARM to a five-stage Xilinx MicroBlaze [12] soft core and a five-stage ARMv2a-compatible Amber 25 [23] soft core. To only compare pipelines, DRAM controllers and other peripherals are excluded and memory sizes are all set to 8 kB. The PTARM and MicroBlaze pipelines are both targeted for 75 MHz on a Xilinx Virtex-5 and Amber 25 pipeline for 40 MHz on a Xilinx Spartan-6. The resource utilization for the pipelines (with and without DSP multiplier units) and DRAM controllers after place and route is shown in Table II. The PTARM DRAM VHDL implementation does not include a DMA unit or refresh mechanism and is not used in the evaluation.

Because of different instruction set architectures, the resource comparisons are just general reference points. They do, however, confirm our conjecture that our repeatable thread-interleaved pipeline, scratchpad and memory controller can lead to similar or fewer resources compared to conventional architectures that use hardware techniques to optimize average-case performance.

B. Performance Evaluation Setup

A performance evaluation of the DRAM controller has been done in Reineke *et al.* [5]. Here we evaluate the performance of the PTARM thread-interleaved pipeline synthesized on FPGA. Additional hardware on the PTARM soft core monitors instruction and cycle counts. We compare the execution of several Mälardalen WCET benchmarks [24] on a PTARM soft core against the SimIT-ARM [25], a cycle-accurate simulator of the StrongARM1100 [26], and an HDL cycle-accurate functional simulation of the Amber 25. The StrongARM1100 contains a five-stage pipeline, branch delay slots without branch prediction, a 16 kB instruction cache and an 8 kB data cache. The StrongARM1100 is implemented with 0.35*um* process technology, and can be clocked from 133 MHz to up to 220 MHz. Amber 25 is similar but targeted as a soft core on FPGA, running at 40 MHz on a Xilinx Spartan-6 or 80 MHz on a Xilinx Virtex-6. The current soft core implementation of PTARM clocks at 75 MHz. Because of achievable clock rates differences between different FPGA and silicon technologies, we use clock cycles as our unit of measurement in our experiments.

ARM cross-compilers are used to compile the benchmarks for all architectures. Due to lack of support for different versions of GCC on the simulators and the architectures supporting different versions of the ARM ISA, ARM GCC version 4.6.1 is used to compile for PTARM, ARM GCC version 3.2 for SimIT-ARM, and ARM GCC version 4.5 for Amber 25. To minimize differences between compilers, no optimization flags are used.

Because the Mälardalen benchmarks are single threaded, we set up our experiments as if the same benchmark was running on all four threads of the PTARM architecture[3], and four times in a row on the SimIT and Amber 25 simulators. This way, the total number of instructions executed on both architectures are roughly the same, and the setup mimics independent tasks or an embarrassingly parallel application.

To remove the impact of scratchpads or caches, the benchmarks fit entirely within the scratchpad of PTARM. To give the appearance of single-cycle memory accesses, we modify the SimIT and Amber 25 simulators to not count cycles during a cache miss. When cache misses are counted, execution of the same program may exhibit significantly different execution times depending on the state of the cache when the program starts.

We compare the instruction throughput, shown in Figure 1, for several benchmarks.

TABLE II
RESOURCE UTILIZATION OF PTARM, MICROBLAZE AND AMBER 25

	LUTs	FFs	DSP Slices	BRAM (in Kb)
PTARM pipeline	1414	1001	3	72
Microblaze pipeline	1496	1130	3	72
PTARM pipeline (no mul)	1380	967	0	72
Microblaze pipeline (no mul)	1503	1090	0	72
Amber 25 pipeline (no mul)	7608	2847	3	448
PTARM DRAM controller	1551	2181	0	2
Microblaze DRAM controller	2175	3049	0	13

C. Results and Analysis

Several observations can be made from these measurements. Most importantly, we observe from Figure 1 that PTARM almost achieves one instruction per cycle throughput for all

[3] To enable a simpler measurement of clock cycles on the FPGA, instructions and thread cycles are counted in hardware on only one specific thread. This thread cycle count is then multiplied by four to get the total count for all threads.

benchmarks. The few multi-cycle instructions, such as load and store multiple, are the only reason it does not achieve a one instruction per cycle throughput. The thread-interleaved pipeline removes the control and data hazards in the pipeline. On the contrary, with the single-threaded StrongARM 1100 and Amber 25, the effects of pipeline hazards reduce the throughput of instructions, as the pipeline needs to stall for control and data hazards that can arise. The higher instruction throughput achieved by interleaving hardware threads in the pipeline comes from trading off single-thread latency. The thread-interleaved pipeline time-shares the pipeline resources between the hardware threads, so the latency of a single thread is higher compared to a single-threaded pipeline. However, the simplified hardware design of thread-interleaved pipelines allows us to clock the pipeline at higher frequencies [20], which can mitigate the single-thread performance loss. Furthermore, for applications with enough parallelism to fully utilize the pipeline, the higher instruction throughput gives better overall performance. The main reason Amber 25 has a lower throughput than StrongARM 1100 is that Amber 25 has an area-efficient but slower multiplier. This, and compiler differences, result in the variations in the instruction counts.

Communication between threads can occur through the scratchpad or by reallocating the DRAM banks from one thread to another. If semaphores or mutexes are used in such communication, then obviously the timing of one thread will affect the timing of the other. But more interestingly, because of our precise control over timing, deterministic communication can occur *without* semaphores or locks, as done in [27]. This leads to extremely low overhead multithreading.

D. Applications

Our experimental setup assumes perfect parallelism and full utilization of scratchpads. Such assumptions are not unrealistic: Liu *et al.* [27] present a real-time engine fuel rail simulator that contains this setup and directly benefits from the improved throughput previously shown. The simulator implements a one-dimensional computational fluid dynamics (1D-CFD) solver that improves the precision of fuel injection, leading to more efficient engine designs. The system progresses in time steps, and the fuel rail is split up into hundreds of pipe segments. Every time step, the pipe segments solve for their pressure and flow rate according to their pipe configuration, and communicate the values to their neighboring nodes. Each pipe segment is mapped to a hardware thread on the PTARM. The code for each pipe segment fits entirely in the scratchpads of PTARM, and communication across PTARMs occurs through single-cycle shared local memory. A time-triggered execution model enforces that this communication only occurs at the end of every time step. Thus, during each time step, the computations for each node are completely independent, similar to our experimental setup in Section IV-B.

Integrated architectures [28], [29] may also benefit from the improved throughput presented in PTARM [4]. Contrary to the conventional *federated architectures*, in which features are implemented on physically separated platforms, *integrated architectures* aim to implement multiple features on a single platform to save resources. As more and more features are packed onto a single platform, the throughput of the architecture is becoming increasingly important. The temporal isolation of the hardware threads in PTARM can ensure that features will not disrupt the temporal properties of each other, and the improved throughput from PTARM can allow for better system performance.

Whether it is *intra-application parallelism*, as in the fuel rail simulator, or *inter-application parallelism*, as in integrated architectures, when the application presents sufficient parallelism, PTARM can improve system performance while maintaining repeatable execution times.

V. Conclusion

In this paper we introduce the Precision-Timed ARM (PTARM) microarchitecture that includes a thread-interleaved pipeline, a scratchpad memory, and a DRAM controller with repeatable access latencies. We evaluate a VHDL implementation of the processor on a Xilinx Virtex 5 FPGA. The benchmarks show that the proposed architecture has competitive throughput for programs that fit the scratchpad and consist of independent tasks or are embarrassingly parallel.

The proposed architecture also introduces new research challenges. By removing the cache, the task of moving data between the scratchpad and the DRAM is shifted from hardware to software. We consider efficient, predictable, and repeatable scratchpad allocation as an interesting direction for future work. Furthermore, programming models for PRET machines—with temporally isolated threads—is another important area worth exploring.

References

[1] E. A. Lee, "Cyber physical systems: Design challenges," in *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. IEEE, 2008, pp. 363 – 369.

[2] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, May 2006.

[3] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl, "A disruptive computer design idea: Architectures with repeatable timing," in *ICCD*. IEEE, October 2009, pp. 54–59.

[4] I. Liu, J. Reineke, and E. A. Lee, "A PRET architecture supporting concurrent programs with composable timing properties," in *44th Asilomar Conference on Signals, Systems, and Computers*, November 2010.

[5] 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*. ACM, October 2011, pp. 99–108.

[6] R. Wilhelm *et al.*, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE TCAD*, vol. 28, no. 7, pp. 966–978, 2009.

[7] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, no. 1, pp. 1–24, 2009.

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

[9] J. Whitham and N. Audsley, "MCGREP - A Predictable Architecture for Embedded Real-time Systems," in *Proc. RTSS*, 2006, pp. 13–24.

[10] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54, no. 1, pp. 265–286, 2008.

[11] S. Andalam, P. Roop, and A. Girault, "Predictable multithreading of embedded applications using PRET-C," in *MEMOCODE*. IEEE, 2010, pp. 159–168.

[12] Xilinx, "MicroBlaze Processor Reference Guide - Embedded Development Kit EDK 13.4," 2012, available from: http://www.xilinx.com. [Last accessed: May 14, 2012].

[13] J. E. Thornton, "Parallel Operation in the CDC 6600," in *AFIPS Proc. FJCC*, 1964, pp. 33–40.

[14] 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*, vol. 35, no. 9, pp. 1320–1333, 1987.

[15] B. Smith, "The architecture of HEP," in *on Parallel MIMD computation: HEP supercomputer and its applications*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1985, pp. 41–55.

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

[17] Parallax propeller chip. Available from: http://www.parallax.com/. [Last accessed: August 16, 2012].

[18] J. Glossner, E. Hokenek, and M. Moudgill, "Multi-threaded processor for software defined radio," in *Software Defined Radio Technical Conference and Product Exposition*, 2002, pp. 195–199.

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

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

[21] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *CODES*. ACM, 2002, pp. 73–78.

[22] Xilinx. (2009, February) Virtex-5 family overview.

[23] OpenCores. Amber arm-compatible core. Available from: http://opencores.org/project,amber. [Last accessed: August 16, 2012].

[24] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks – past, present and future," in *WCET*, B. Lisper, Ed. Brussels, Belgium: OCG, July 2010, pp. 137–147.

[25] W. Qin and S. Malik, "Flexible and formal modeling of microprocessors with application to retargetable simulation," in *DATE*. Washington, DC, USA: IEEE, 2003, pp. 556–561.

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

[27] I. Liu, E. A. Lee, M. Viele, G. G. Wang, and H. Andrade, "A heterogeneous architecture for evaluating real-time one-dimensional computational fluid dynamics on FPGAs," in *IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM), Toronto, Canada*, April 2012.

[28] C. Watkins and R. Walter, "Transitioning from federated avionics architectures to integrated modular avionics," *26th Digital Avionic Conference*, October 2007.

[29] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "From a federated to an integrated automotive architecture," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 28, no. 7, pp. 956–965, 2009.