

Reconciling Repeatable Timing with Pipelining and Memory Hierarchy ^{*}

Stephen A. Edwards¹, Sungjun Kim¹, Edward A. Lee²,
Hiren D. Patel², and Martin Schoeberl³

¹ Columbia University, New York, NY, USA,
{sedwards, skim}@cs.columbia.edu

² UC Berkeley, Berkeley, CA, USA,
{eal, hiren}@eecs.berkeley.edu

³ Vienna University of Technology, Vienna, Austria
mschoebe@mail.tuwien.ac.at

Abstract. This paper argues that repeatable timing is more important and more achievable than predictable timing. It describes microarchitecture approaches to pipelining and memory hierarchy that deliver repeatable timing and promise comparable or better performance compared to established techniques. Specifically, threads are interleaved in a pipeline to eliminate pipeline hazards, and a hierarchical memory architecture is outlined that hides memory latencies.

1 Introduction

A conventional microprocessor executes a sequence of instructions from an instruction set. Each instruction in the instruction set changes the state of the processor in a well-defined way. The microprocessor provides a strong guarantee about its behavior: if you insert in the sequence an instruction that observes the state of the processor (e.g., the contents of a register or memory), then that instruction observes a state equivalent to one produced by a sequential execution of exactly every instruction that preceded it in the sequence. For speed, however, modern microprocessors rarely execute the instructions strictly in sequence. Instead, pipelines, caches, write buffers, and out-of-order execution reorder and overlap operations while preserving the illusion of sequential execution. Any *correct execution* must preserve the strong guarantee, and thus the illusion.

Because the semantics of sequential instruction execution is specified precisely at the bit level, the state observed by a particular instruction is *repeatable*, meaning that every correct execution of the same sequence will lead to the same state given the same inputs. If the sequence of instructions is that given by a single program specifying a

^{*} This work was supported by the National Science Foundation (NSF award #0720882 (CSREHS: PRET) and the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives additional support from the U. S. Army Research Office (ARO #W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, Lockheed-Martin, National Instruments, Thales, and Toyota.

computation whose inputs are included in the initial state of the processor (e.g. in memory) and whose outputs are included in the final state, then the behavior of the program is repeatable. We call this a conventional Turing-Church computation.

Very few instruction sets provide any guarantee about the *timing* of the execution of a sequence of instructions. If the sequence of instructions is specifying a conventional Turing-Church computation, then this timing is irrelevant. The sequence specifies a mapping from inputs (contained in the initial state of the processor) to outputs (contained in the observed state of the processor).

For many applications, and most particularly for embedded systems, the timing does matter, however. In particular, some instructions in the sequence specify interactions with the external physical world, causing actuation of physical devices for example. Some will poll sensors that measure the state of the physical world at the time the instruction is executed. Some instructions will be inserted into the sequence in response to an external physical event that raises an interrupt request. The time at which this occurs determines where in the sequence the instructions to service the interrupt are inserted. Thus, the sequence of instructions executed by the microprocessor is not entirely determined by a program, but is also affected by the timing of external events. Even non-embedded computations will use such interrupts to perform multitasking, executing multiple threads concurrently and switching between them in response to interrupts raised by an external timer or external devices such as disk drives. Again, the sequence of instructions is not completely specified by the program(s) being executed. Hence, the strong guarantee provided by the microprocessor is not sufficient to make the behavior of the programs repeatable.

For such programs, the inputs to the system are not just the initial state of the processor, as they are in a conventional Turing-Church computation. Any complete definition of “inputs” must include the timing of interrupts and the time at which sensor values are polled. Any complete definition of “outputs” must also include the timing at which actuations in the physical environment are asserted. These clearly affect the behavior of the system. For a microprocessor that provides no timing guarantees, *no such program has repeatable behavior*. Two “correct” executions can exhibit significantly different timing and can execute significantly different sequences of instructions, resulting in significantly different outputs.

In the above analysis, we implicitly define the *behavior* of a program to be the mapping from inputs to outputs. Many useful programs, however, do not require such a rigorously defined behavior. Some measure of nondeterminism is tolerable, meaning that the same inputs may lead to different outputs, as long as some application-dependent set of *properties* is satisfied. If the timing of an output is important, for example, it may not have to be precise. The application has some tolerance to deviations in the timing. Thus, we are generally more interested in whether satisfaction of these properties is repeatable. That is, we insist that every correct execution satisfies an application-dependent set of properties.

A real-time program, for example, will specify a set of properties as constraints on the timing of certain external interactions or internal actions (updates of values in memory, for example). The task of real-time system designer is to ensure that these properties are repeatable.

A *predictable* property is a repeatable property that can be determined in finite time from a specification of the system. Since any computer only has finite memory, the state after a sequence of instruction executions is technically predictable, although doing so can take an impractically long time. However, if the specification of the system is a program, the sequence of instructions executed will not be predictable if timing is not repeatable (interrupts and multitasking will interfere). Thus, even a conventional Turing-Church computation on a uniprocessor may not have repeatable behavior [1].

Researchers have made great strides in predicting execution time [2, 3], specifically in *bounding* the execution time, determining worst-case execution time (WCET). However, existing techniques can only determine WCET for a processor-program pair, not for just a program (unlike processor state, which must be consistent across all correct processors). Even worse, implementation details that can affect execution time, such as memory consistency models [4], are often not well-specified. Researchers are calling for moderation and identifying particularly problematic techniques [5–7].

Moderating these practices is not enough. Repeatability is more important than predictability. With repeatable timing, testing can establish correctness, and testing is almost always easier than detailed analysis. Without repeatability, testing proves little.

Timing should be a repeatable property of a *program*, not of a program executing on a particular processor implementation. That is, our notion of “correct” execution of a sequence of instructions should include timing properties. This requires changes to the semantics of instruction sets.

A few researchers have addressed the problem of repeatable timing. Precision-timed (PRET) machines [8, 9] modify the instruction set for repeatable timing. Mueller’s VISA [10] runs a standard fast processor in concert with a slow (repeatable) one, switching over if the fast one lags behind. Schoeberl has implemented a Java processor where time-repeatability of individual bytecode instructions was the major design goal [11]. Whitham and Audsley’s MCGREP [12] use programmable microcode to accelerate hotspots that are otherwise too slow.

In this paper, we focus on two intertwined obstacles to repeatable timing: pipelines and memory hierarchy. We show that repeatable timing can be reconciled with pipelining and memory hierarchy, both of which are required to get competitive performance.

2 Pipeline Interleaving

Pipelining improves hardware performance: instead of waiting for every operation in an instruction to complete before starting the next instruction, start the second instruction while the first instruction completes. The challenge comes when successive instructions affect each other, such as when an instruction controls a branch immediately following it or writes to a register that is read by the following instruction. Dealing with these hazards requires additional control and steering logic and makes the execution time of an instruction depend on the instructions surrounding it.

Instead of rejecting pipelining outright, we advocate an interleaved pipeline, a form of fine-grained multithreading [13] (also known as interleaved multithreading). In every cycle, an instruction from a different thread is fetched and inserted into the pipeline, provided the thread’s previous instruction has completed (i.e., some instructions may

take multiple cycles). Thus, at any time, the pipeline is running at most one instruction from each thread. From the perspective of each thread, there is no pipeline; each instruction completes before the next one begins.

Interleaved pipelines have performance advantages. They eliminate inter-instruction dependencies, eliminating time-consuming hazard detection and steering. They also reduce the off-chip memory latency penalty because other threads execute while one is waiting for memory. This technique has been used in various research and commercial processors for achieving higher performance since the early 80s [13].

More importantly, pipeline interleaving leads to repeatable timing [14]. By removing the danger of inter-pipeline control and data dependencies, most instructions can take the same number of cycles to execute every time they enter the pipeline. Instructions that block (e.g., when accessing memory) can always block for the same number of cycles, regardless of what is happening in other threads and which instructions precede or follow them.

The first PRET machine [9] implements a thread-interleaved pipeline with each thread having its own thread context. The memory hierarchy consists of a scratchpad memory shared by all threads and a memory wheel component that arbitrates access to the main memory in a time-triggered fashion. A replay technique is used for any instructions that take multiple cycles, such as main memory accesses. A novel concept in PRET is the ability to control temporal behaviors in software through deadline instructions [15, 9]. The combination of the architecture and the deadline instructions enables programmers to get repeatable timing.

3 Memory Hierarchy

While memory bandwidth can be improved with a host of tricks (mainly parallelism), memory latency is a fundamental problem for large memories. The usual solution is a hierarchy: a mix of large slow memories feeding small, fast ones. Standard memory hierarchies use caching to preserve the illusion of a large, undifferentiated memory. While caches present the programmer with a convenient abstraction, they leave timing unpredictable and often non-repeatable [7]. For a program running in isolation, the time taken for a memory access depends on which cache it resides in. This depends in part on its address, which is often difficult to predict before the program is running, but also on the history of the memory accesses.

Our solution retains the memory hierarchy but manages it differently. Large modern DRAM chips are well-suited to our interleaved pipeline. Internally, they consist of separately operating banks (e.g., eight) that can be simultaneously at various stages of a read or write. Our solution is to assign threads to separate banks. Since the threads are interleaved in the pipeline, we can effectively hide the memory latency. In each cycle, a thread is granted exclusive access to its memory bank and may initiate or continue a memory operation. Like the interleaved pipeline, the memory scheduler will implement something like a round-robin policy.

Combining this with an SRAM scratchpad memory that is shared among the threads, we note that our memory hierarchy is the converse of a conventional multicore approach. The fast, close memory is shared among concurrent threads, while the slow,

remote memory is private to each thread. In many multicore architectures, the fast, close memory is a private cache, and the slow, remote memory is shared.

This architecture suggests numerous interesting possibilities that have profound implications on the programming models for concurrency. For example, one could dynamically (but infrequently) change the ownership of memory banks to transfer large quantities of data among threads, while using the smaller, shared scratchpad SRAM for synchronization and fine-grain coordination. One could also vary how banks are assigned to threads. Granting a thread exclusive access to a bank will lead to the highest performance, but it will also be possible to share a bank among multiple threads through a secondary round-robin schedule and still achieve repeatable timing. Moreover, DRAM refresh, which usually disturbs predictable timing, can be triggered during instructions that do not access memory (e.g., branches). If a basic block is too large to guarantee enough refresh cycles, nop instructions can be inserted by the compiler to trigger the refresh. This still achieves repeatable timing (however, prediction may be harder).

Sharing memory banks by supplying a periodic schedule is a time-triggered approach, which has been used successfully for networking, but not memory access. Pitter and Schoeberl [16] have also considered memory access (in their case, DMA) as a real-time scheduling problem, but treat it as a more traditional real-time task and worry just about WCET. Rosen et al. [17] similarly consider bus access as a real-time task.

4 Discussion

To have high processor utilization, our approach requires that application developers expose enough concurrency that multiple threads can be active much of the time. This suggests our architecture may be better used with programming models that are intrinsically concurrent. Fortunately, there is a great deal of momentum towards such programming models, particularly for the design of embedded real-time systems. Commercial tools such as Simulink with Real-Time Workshop from The MathWorks, TargetLink from dSpace, and LabVIEW from National Instruments, all provide intrinsically concurrent programming models and synthesize concurrent embedded code. Emerging programming models for real-time systems like Giotto [18], TDL [19], and Prides [20] also expose a great deal of exploitable concurrency, and appear to be good matches for our architecture. Even traditional RTOS-based designs [21] can benefit from our approach because the real-time constraints will be easier to guarantee with concurrency that delivers repeatable timing.

References

1. Lee, E.A.: The problem with threads. *Computer* **39**(5) (2006) 33–42
<http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html>.
2. Thiele, L., Wilhelm, R.: Design for timing predictability. *Real-Time Systems* **28**(2-3) (2004) 157–177
3. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.* **7**(3) (2008) 1–53

4. Hill, M.D.: Multiprocessors should support simple memory-consistency models. *IEEE Computer* **31**(8) (August 1998) 28–34
5. Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., Ferdinand, C.: Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems* **28**(7) (2009)
6. Kirner, R., Puschner, P.: Obstacles in worst-case execution time analysis. In: *Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, Orlando, FL, USA, IEEE (2008) 333–339
7. Schoeberl, M.: Time-predictable computer architecture. *EURASIP Journal on Embedded Systems* **2009**(Article ID 758480) (2009) 17 pages
8. Edwards, S.A., Lee, E.A.: The case for the precision timed (PRET) machine. In: *Design Automation Conference (DAC)*, San Diego, CA (2007)
9. Lickly, B., Liu, I., Kim, S., Patel, H.D., Edwards, S.A., Lee, E.A.: Predictable programming on a precision timed architecture. In Altman, E.R., ed.: *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, Atlanta, GA, USA, ACM (October 2008) 137–146
10. Anantaraman, A., Seth, K., Patil, K., Rotenberg, E., Mueller, F.: Virtual simple architecture (visa): exceeding the complexity limit in safe real-time systems. In: *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*. Volume 31, 2 of *Computer Architecture News.*, New York, ACM Press (June 9–11 2003) 350–361
11. Schoeberl, M.: A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture* **54**/1–2 (2008) 265–286
12. Whitham, J., Audsley, N.: MCGREP - A Predictable Architecture for Embedded Real-time Systems. In: *Proc. RTSS*. (2006) 13–24
13. Ungerer, T., Robič, B., Šilc, J.: A survey of processors with explicit multithreading. *Computing Surveys* **35**(1) (2003) 29–63
14. Lee, E.A., Messerschmitt, D.G.: Pipeline interleaved programmable dsps: Architecture. *IEEE Trans. on Acoustics, Speech, and Signal Processing* **ASSP-35**(9) (1987)
15. Ip, N.J.H., Edwards, S.A.: A processor extension for cycle-accurate real-time software. In: *IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*. Volume LNCS 4096., Seoul, Korea, Springer (2006) 449–458
16. Pitter, C., Schoeberl, M.: Time predictable CPU and DMA shared memory access. In: *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands (August 2007) 317 – 322
17. Rosen, J., Andrei, A., Eles, P., Peng, Z.: Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In: *Proceedings of the Real-Time Systems Symposium (RTSS 2007)*. (Dec. 2007) 49–60
18. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: *EMSOFT 2001*. Volume LNCS 2211., Tahoe City, CA, Springer-Verlag (2001) 166–184
19. Pree, W., Templ, J.: Modeling with the timing definition language (tdl). In: *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*. LNCS, San Diego, CA, Springer (2006)
20. Zhao, Y., Lee, E.A., Liu, J.: A programming model for time-synchronized distributed real-time systems. In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Bellevue, WA, USA, IEEE (2007) <http://ptolemy.eecs.berkeley.edu/publications/papers/07/RTAS/>.
21. Buttazzo, G.C.: *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. second edn. Springer (2005)