A Disruptive Computer Design Idea: Architectures with Repeatable Timing

Stephen A. Edwards
Coumbia University
sedwards@cs.columbia.edu

Sungjun Kim
Coumbia University
skim@cs.columbia.edu

Edward A. Lee
UC Berkeley
eal@eecs.berkeley.edu

Isaac Liu
UC Berkeley
liuisaac@eecs.berkeley.edu

Hiren D. Patel
UC Berkeley
hiren@eecs.berkeley.edu

Martin Schoeberl
Vienna University of Technology
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.**

I. 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 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 application, 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

This work was supported by the National Science Foundation (NSF award #0720882 (CSR-EHS: 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.

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

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 a real-time system designer is to ensure that these properties are repeatable.

A *predictable* property is a repeatable property than 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], [6], [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.

II. 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 depend on each other, such as the instruction following a conditional branch or writes to a register that is read by the following instruction. Dealing with these hazards requires additional control and steering logic to reorder or stall the instruction, which 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 in a round-robin fashion. Instead of stalling the pipeline, instructions that take multiple cycles (i.e., memory access instructions) will be re-fetched into the pipeline on the next round-robin cycle, until the instruction is finished. Thus, at any time, the pipeline is running at most one instruction from each thread when there are more threads than pipeline stages. 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 hide 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 data dependencies and hazards in the pipeline, instructions will never be affected by their surrounding instructions. Each instruction will now take exactly the same number of cycles to execute each time it enters 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

« The first memory access request arrives here — The next memory access request cannot be processed before this time »

Clock No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Command	ACT	READ	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	ACT	R/W	NOP	NOP
Bank Address	00	00	X	X	X	X	X	X	X	X	X	00	00	X	X
Address	Row	Col	X	X	X	X	X	X	X	X	X	Row	Col	X	X
Bank 0		AL	AL	CL	CL	CL	Data 0	Data 0	tRP	tRP	tRP		AL	AL	CL
Data I/O							Data 0	Data 0							

(a) Read followed by R/W request of DDR2: CPU requests read to a row in a bank and then read/write to a different row in the same bank. Additive latency(AL) = 2, column latency(CL) = 3, burst length(BL) = 4, row precharge time(tRP) = 3, auto-precharge.

« The first memory access request arrives here — The next memory access request cannot be processed before this time »

Clock No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Command	ACT	WRITE	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	ACT	R/W
Bank Address	00	00	X	X	X	X	X	X	X	X	X	X	X	00	00
Address	Row	Col	X	X	X	X	X	X	X	X	X	X	X	Row	Col
Bank 0		AL	AL	CL	CL	Data 0	Data 0	tWR	tWR	tWR	tRP	tRP	tRP		AL
Data I/O						Data 0	Data 0								

(b) Write followed by R/W request of DDR2: CPU requests write to a row in a bank and then read/write to a different row in the same bank. Additive latency(AL) = 2, column latency(CL) = 3, burst length(BL) = 4, write recovery time(tWR) = 3, row precharge time(tRP) = 3, auto-precharge.

Fig. 1. Timing of read and write memory operation.

architecture and the deadline instructions enables programmers to get repeatable timing.

III. 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. First, we use scratchpad memories (SPMs) instead of caches. SPMs use less power and occupy less area than caches because no speculation logic is needed. Compared to caches, SPMs give us the same access time, except that the allocation of data on the SPMs are done in software, instead of hardware. This allows us to gain repeatable performance in the fast access memory.

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 accessed at various stages of a read or write. Our solution is to assign threads to dedicated banks. Since the threads are interleaved in the pipeline, we can effectively hide the memory latency. In each cycle, a thread is granted access to its dedicated 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.

A. Making DRAM Accesses Repeatable

We design a DRAM memory controller that guarantees repeatable timing behaviors when accessing the DRAM memories. This design is tightly coupled to the interleaved pipeline to hide the long latencies of DRAM, and consequently improve performance. We use the Samsung specifications [16], [17], [18], [19] for our DDR2 memory.

Figure 1(a) and Figure 1(b) illustrate the basic read and write memory operations on the same bank but on different rows of the DDR2 memory. In these figures, the signals driven or read by the controller are denoted by command, bank address, address, and data I/O. Every cell in the figure represents the signal value at a particular clock cycle. For instance, at clock cycle 1 in Figure 1(a), the bank address selects bank 0, the command to activate (ACT) the bank is sent, and the row address is supplied.

Figure 1(a) shows that a read operation emits the requested data on the seventh and eighth cycles, but it requires an additional three cycles for precharge amounting to a total of 11 clock cycles before the next memory access command can begin. The write operation in Figure 1(b) shows that data is written to the memory on the sixth and seventh cycle of the write operation, but due to write recovery and precharge times, the next memory operation can only occur after 13 cycles. Notice that Figure 1(a) and Figure 1(b) show the memory operations with the longest latencies for a read and write. Other combinations of memory operations yield different timing behaviors. For example, consecutively reading from the same bank and the same row results in a shorter memory access latency while writing then reading from the same bank and same row result in a different access latency. Because of such variations, the history of memory operations affects the latency of the next memory operation. This results in non-repeatable timing behaviors when using conventional DRAM memory controllers.

« The initially scheduled thread requests memory access at this time

« The secondly scheduled thread can request memory access from this time

The initial thread can request memory access after this time again »

	Clock No.	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	Command	ACT	R/W	NOP	ACT	R/W	NOP	NOP	NOP	NOP	NOP	NOP	NOP	NOP	ACT	R/W
	Bank Address	00	00	X	01	01	X	X	X	X	X	X	X	X	00	00
	Address	Row	Col	X	Row	Col	X	X	X	X	X	X	X	X	Row	Col
Bank 0	read case		AL	AL	CL	CL	CL	Data 0	Data 0	tRP	tRP	tRP				AL
	write case		AL	AL	CL	CL	Data 0	Data 0	tWR	tWR	tWR	tRP	tRP	tRP		AL
Bank 1	read case					AL	AL	CL	CL	CL	Data 1	Data 1	tRP	tRP	tRP	
	write case					AL	AL	CL	CL	Data 1	Data 1	tWR	tWR	tWR	tRP	tRP
	Data I/O range						Data 0	Data 0	Data 0	Data 1	Data 1	Data 1				

Fig. 2. Interleaved pipeline operation of DDR2 (our design approach): CPU requests read/write to a row in a bank and then read/write to a row in another bank. Additive latency(AL) = 2, column latency(CL) = 3, burst length(BL) = 4, write recovery time(tWR) = 3, row precharge time(tRP) = 3, auto-precharge

One approach in achieving repeatability tightly couples the processor's interleaved pipeline with the memory controller. It also designates each DRAM bank to a hardware thread. Since the interleaved pipeline schedules each hardware thread using a round-robin policy, this directly translates to the DRAM such that the memory access order also follows round-robin scheduling among banks. In Figure 2, we show how to exploit the interleaved pipeline while achieving repeatability. (In this diagram, we only show two threads, thus showing only two memory banks.)

As can be seen, a thread can access the memory again every 13 cycles, but the next thread is also able to access it every 13 cycles because of the DRAM's ability to access separate banks simultaneously. To be specific, no matter what the command types are received to a bank, the next access is allowed after the maximum latency (the write latency) if it is to a separate bank. Since a different thread is scheduled every cycle in an interleaved pipeline, and a different bank is used for every successive memory access, we can interleave memory accesses of different banks such that all memory operations take 13 cycles. In this way, we get repeatable timing behaviors from memory operations.

In addition, compared with Figure 1(a) and Figure 1(b) where only one memory operation at a time is possible, interleaving as shown in Figure 2 improves the throughput of memory operations. While the next memory operation is processed after 11 cycles in Figure 1(a) and 13 cycles in Figure 1(b), the subsequent memory operations can be processed after 3 cycles or 10 cycles in Figure 2, or after 6 cycles or 7 cycles provided we evenly distribute the access scheduling. Furthermore, these latencies can be reduced with more threads/banks (i.e., 3 and 4 cycles with 4 threads/banks provided we add the memory access timing every 3 cycles).

Note that in order to maintain repeatability, we need arbitration between the processor and the memory controller because they might run at different clock speeds. As a result, we add the maximum interaction latency to the total memory access latency and force the processor's memory access instruction to always take this total amount of time. Scheduling of the memory control is optimized to reduce the interaction latency.

B. DRAM Refreshing

Another source of non-repeatability is the constant need to refresh the DRAM. When the DRAM is being refreshed, the processor cannot access the DRAM. Conventional processors simply stall until the refresh is completed. However, this is undesirable.

In our approach, we propose using the distributed, RAS-only refresh [20] to each bank separately. In other words, memory refresh is equivalent to a row access to a bank; thus, each bank can refresh its row at the proper time separately. Instead of allowing the internal memory logic to refresh all the memory banks at once [18], we tightly couple the bank refreshes with the processor hardware threads, and bring this abstraction up to the software. When an instruction does not access memory, the memory controller triggers DRAM refreshing. Note that since each thread has its dedicated bank, this will not cause any timing variations among threads.

When a refresh is required can be statically analyzed. For example, we can issue a refresh whenever we encounter a branch or nop instruction. Thus, provided a basic block takes less time than the required refresh rate (usually 15.6 μs), then stable DRAM refresh is guaranteed because the basic block will finally meet a branch instruction that triggers refresh. On the other hand, provided a basic block is too large, then the compiler can insert a nop within that the basic block to guarantee that the DRAM is refreshed. This still achieves repeatable timing (however, prediction may be harder).

C. Shared Memory

Combining the DRAM controller 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.

Furthermore, 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 [21] 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. [22] similarly consider bus access as a real-time task.

D. Programmable DMA Controller

To facilitate fast transfer of program code and data between the fast SRAM and large DRAM memories, we advocate using a programmable direct memory access (DMA) controller. Input to the DMA controller is a program that describes the memory transfer pattern for the particular target application. This program is a finite state machine where the states consist of instructions to perform the actual transfer between the memories, and the transitions take place based on the program counter of the hardware threads. Therefore, the DMA controller has also access to the address bus on which it can snoop the program counter of the hardware threads. This approach has the advantage that the program dictating memory transfers can execute in parallel with the target application. It also does not require the main application program to explicitly issue a transfer instruction to initiate the transfer. Instead, it is possible for the programmable DMA to begin transferring code and data before the application requires it. Ideally, the program for the controller will be automatically generated from the application model or code.

E. Split Read Access

A further optimization to hide memory access latencies on a load instruction is possible by splitting the read command phase and the result return phase in the processor. Therefore, a read start instruction just communicates the address and read request to the memory subsystem and returns immediately. To obtain the actual read value another instruction has to be issued. The benefit of the split of the load instruction is that other instructions, which do not access the main memory, can be scheduled by the compiler to hide the access latency. A split read operation is similar to a prefetch instruction, which gives the cache system a hint which data will be used in the near future.

IV. 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 [23], TDL [24], and Ptides [25] also expose a great deal of exploitable concurrency, and appear to be good matches for our architecture. Even traditional RTOS-based designs [26] can benefit from our approach because the real-time constraints will be easier to guarantee with concurrency that delivers repeatable timing.

Along with the concurrency that these programming models provide, the structure that comes with the programming models could also help us analyze memory access patterns and generate appropriate programs for the programmable DMA. For example, Bandyopadhyay [27] used the structured properties of Heterochronous Dataflow programming model to statically analyze when data should be moved between the main memory and a software controlled scratchpad in order to achieve the most optimal allocation scheme. In a similar fashion, we can configure the memory controller and partitioning of shared memory regions.

A challenging issue that arises is how to assess the performance of resulting computer architectures compared to established approaches. Standard benchmarks have no concurrency, and therefore an interleaved pipeline immediately appears to come with an enormous cost. Suppose for example that we compare a non-interleaved processor with a 200 MHz instruction issue rate to an interleaved processor with a 240 MHz instruction issue rate (we assume that an interleaved pipeline can be clocked slightly faster because there is less hardware for forwarding and pipeline interlocks, which often form the critical path that determines the clock rate). Suppose then that a four-way interleaved pipeline is required to remove all pipeline hazards. Then a given thread will execute at an instruction issue rate of 60 MHz (240/4). Suppose that the 200 MHz non-interleaved processor results in an average issue rate of 180 MHz after stalls due to pipeline hazards (probably generous). Then a single thread benchmark will appear to indicate a performance of only 33% of the non-interleaved pipeline. However, four such benchmarks running simultaneously will exhibit a performance that is 133% of the non-interleaved pipeline, a dramatic improvement. Which comparison is more fair?

V. Conclusion

For future embedded systems we need new computer architectures to support the tight integration of computing systems with the surrounding physical world. In this paper we argued that repeatable timing is more important and more achievable than predictable timing. Moreover, we can achieve both repeatable timing and performance. To explore the speedup of pipelining within an architecture with repeatable timing we propose to use pipeline interleaving within the processor pipeline and pipelining of the DRAM access. In the proposed architecture, each hardware supported thread owns one bank of

the DRAM. Communication between the threads is performed on a shared on-chip memory. Therefore, we end up with an inverse memory hierarchy compared with standard architecture: fast on-chip memory is shared, slow off-chip memory is thread local. Future work will explore computing and communication models that fit this architecture.

REFERENCES

[1] E. A. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006, http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html.

[2] L. Thiele and R. Wilhelm, "Design for timing predictability," *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.

[3] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, "The worst-case execution time problem – overview of methods and survey of tools," *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.

[4] M. D. Hill, "Multiprocessors should support simple memory-consistency models," *IEEE Computer*, vol. 31, no. 8, pp. 28–34, Aug. 1998.

[5] R. Wilhelm, D. Grund, J. Reineke, M. Schlickling, M. Pister, and C. Ferdinand, "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 28, no. 7, 2009.

[6] R. Kirner and P. Puschner, "Obstacles in worst-case execution time analysis," in *Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*. Orlando, FL, USA: IEEE, 2008, pp. 333–339.

[7] M. Schoeberl, "Time-predictable computer architecture," *EURASIP Journal on Embedded Systems*, vol. 2009, no. Article ID 758480, p. 17 pages, 2009.

[8] S. A. Edwards and E. A. Lee, "The case for the precision timed (PRET) machine," in *Design Automation Conference (DAC)*, San Diego, CA, 2007.

[9] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable programming on a precision timed architecture," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, E. R. Altman, Ed. Atlanta, GA, USA: ACM, October 2008, pp. 137–146.

[10] A. Anantaraman, K. Seth, K. Patil, E. Rotenberg, and F. Mueller, "Virtual simple architecture (visa): exceeding the complexity limit in safe real-time systems," in *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, ser. Computer Architecture News, vol. 31, 2. New York: ACM Press, June 9–11 2003, pp. 350–361.

[11] M. Schoeberl, "A Java processor architecture for embedded real-time systems," *Journal of Systems Architecture*, vol. 54/1–2, pp. 265–286, 2008.

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

[13] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *Computing Surveys*, vol. 35, no. 1, pp. 29–63, 2003.

[14] E. A. Lee and D. G. Messerschmitt, "Pipeline interleaved programmable dsps: Architecture," *IEEE Trans. on Acoustics, Speech, and Signal Processing*, vol. ASSP-35, no. 9, 1987.

[15] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," in *IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. LNCS 4096. Seoul, Korea: Springer, 2006, pp. 449–458.

[16] Samsung Electronics, Co., "DDR2 SDRAM device operating and timing diagram," 2007, http://www.samsung.com/global/business/semiconductor/products/dram/downloads/ddr2_device_operation_timing_diagram_may_07.pdf.

[17] ——, "Application note: tWR (write recovery time)," 2001, http://www.samsung.com/global/business/semiconductor/products/dram/dram/downloads/applicationnote/tWR.pdf.

[18] ——, "DDR2 SDRAM tRFC application note," 2004, http://www.samsung.com/global/business/semiconductor/products/dram/downloads/applicationnote/app_note_trfc_20040506.pdf.

[19] ——, "DDR2 SDRAM product guide," 2009, http://www.samsung.com/global/business/semiconductor/products/dram/downloads/ddr2_product_guide_may_09.pdf.

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

[21] C. Pitter and M. Schoeberl, "Time predictable CPU and DMA shared memory access," in *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007, pp. 317 – 322.

[22] J. Rosen, A. Andrei, P. Eles, and Z. Peng, "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, pp. 49–60.

[23] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, "Giotto: A time-triggered language for embedded programming," in *EMSOFT 2001*, vol. LNCS 2211. Tahoe City, CA: Springer-Verlag, 2001, pp. 166–184.

[24] W. Pree and J. Templ, "Modeling with the timing definition language (tdl)," in *Automotive Software Workshop San Diego (ASWSD) on Model-Driven Development of Reliable Automotive Services*, ser. LNCS. San Diego, CA: Springer, 2006.

[25] Y. Zhao, E. A. Lee, and J. Liu, "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/.

[26] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd ed. Springer, 2005.

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