Predictable Programming on a Precision Timed Architecture

Ben Lickly
Electrical Engineering and Computer Science
University of California, Berkeley,
Berkeley, CA 94720, USA.
blickly@eecs.berkeley.edu

Isaac Liu
Electrical Engineering and Computer Science
University of California, Berkeley,
Berkeley, CA 94720, USA.
liuisaac@berkeley.edu

Sungjun Kim
Department of Computer Science
Columbia University
New York, NY 10027, USA.
sk3062@columbia.edu

Hiren D. Patel
Electrical Engineering and Computer Science
University of California, Berkeley,
Berkeley, CA 94720, USA.
hiren@eecs.berkeley.edu

Stephen A. Edwards
Department of Computer Science
Columbia University
New York, NY 10027, USA.
sedwards@cs.columbia.edu

Edward A. Lee
Electrical Engineering and Computer Science
University of California, Berkeley,
Berkeley, CA 94720, USA.
eal@eecs.berkeley.edu

ABSTRACT

In a hard real-time embedded system, the time at which a result is computed is as important as the result itself. Modern processors go to extreme lengths to ensure their function is predictable, but have abandoned predictable timing in favor of average-case performance. Real-time operating systems provide timing-aware scheduling policies, but without precise worst-case execution time bounds they cannot provide guarantees.

We describe an alternative in this paper: a SPARC-based processor with predictable timing and instruction-set extensions that provide precise timing control. Its pipeline executes multiple, independent hardware threads to avoid costly, unpredictable bypassing, and its exposed memory hierarchy provides predictable latency. We demonstrate the effectiveness of this precision-timed (PRET) architecture through example applications running in simulation.

Categories and Subject Descriptors

C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Other Architectures Styles*;
C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms

Design

Keywords

Timing Predictability, Memory Hierarchy, Pipeline

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

1. INTRODUCTION

Developing hard real-time software on modern processors has grown very difficult because their complexity has made predicting execution speed nearly impossible. Between multi-stage pipelines with hazards, bypassing, and complex interactions between related instructions, superscalar out-of-order instruction fetch units that almost recompile the program on-the-fly, three-level memory hierarchies with complex cache replacement policies, it is extremely difficult to accurately predict exactly how many cycles it will take to execute a sequence of simple instructions [10], let alone code with conditional branches. Modern processors are truly chaotic [6].

Unfortunately, worst-case execution time bounds are the foundation on which all real-time software engineering is built. Of course, one can always be conservative with over-estimates, but this has become unrealistic since the difference between hitting level one cache and main memory can be a thousand cycles.

We believe the solution for real-time embedded software is no less than a rethinking of processor architecture. As has been argued elsewhere [9], it is time to consider architectures that provide timing as predictable as their function. In this paper, we propose a concrete example of such a precision-timed (PRET) architectures: a multithreaded processor based on the SPARC instruction set architecture (ISA) that delivers predictable timing along with predictable function and performance. Below, we present a cycle-accurate model of the PRET architecture using SystemC [23] and an application running on it to demonstrate how software can take advantage of PRET architectures. In the future, we plan an FPGA implementation as well.

2. THE PRET PHILOSOPHY

The philosophy behind PRET [9] is that modern processor architecture has gone down an unpredictability hole due to its single-minded focus on average-case performance. It needs to be rethought to be effective for real-time embedded systems. Patterson and Ditzel's similar observation [25] started the RISC revolution. In the same way, we must rethink real-time embedded processor architectures.

The complexity of modern processors [13] has made the task of calculating or even bounding the execution time of a sequence of

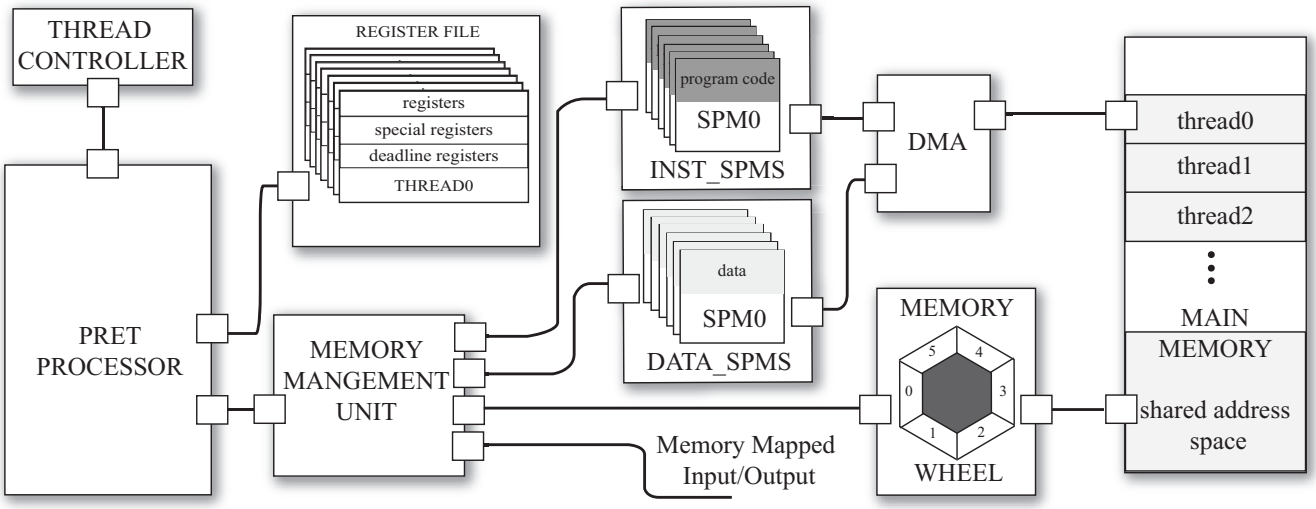


Figure 1: Block Diagram of PRET Architecture

operations very difficult [10]. While this is not critical for best-effort computing, it is a disaster for hard real-time systems.

The PRET philosophy is that temporal characteristics should be as predictable as function. Much like how arithmetic on a processor is always consistent, predictable, and documented, we want its speed to be equally consistent, predictable, and documented. While turning the clock back to the era of eight-bit microprocessors is one obvious way to achieve this, instead the goal of PRET is to re-think many of the architectural features enabled by rising integration levels and render them predictable.

Thus, PRET espouses software-managed scratchpad memories [3], thread-interleaved pipelines with no bypassing [20, 11], explicit timing control at the ISA level [15], time-triggered communication [17] with global time synchronization [16], and high-level languages with explicit timing [14]. In this paper, we propose an architecture that embodies many of these tenants, and demonstrate how it can be programmed. In particular, we focus on integrating timing instructions to a thread-interleaved pipeline and a predictable memory system. We then show how to program such a predictable architecture.

3. RELATED WORK

The Raw processor of Agarwal et al. [29] shares certain elements of the PRET philosophy. It, too, employs a software-managed scratchpad instead of an instruction cache [22], and definitely takes communication delay into account (the name "raw" is a reminder that its ISA is exposed to wire delay). However, it sports multiple single-threaded pipelines with bypassing, a fairly traditional data cache, and focuses almost purely on performance, as usual at the expense of predictability.

The Raw architecture is designed as a grid of single-threaded processors connected by a novel interconnection network. While we envision a similar configuration for high-end PRET processors, the implementation we present here does not consider inter-core communication. We may adopt a Raw-like communication network in the future.

The Java Optimized Processor [27] enables accurate worst-case execution time bounds, but does not provide support for controlling execution time. The SPEAR [8] processor prohibits conditional branches, which we find overly restrictive. The REMIC [26] and KIEL [21] are predictable in the PRET sense, but they only allow Esterel [5] as an entry language. Again, we find this overly restrictive; a central goal of our work was to provide a C development environment.

Ip and Edwards [15] first implemented the deadline instruction in a very simple non-pipelined processor that did not have C compiler support. This deadline instruction allowed a programmable method to specify the lower bound execution time on segments of program code. Our work extends theirs to a new architecture by borrowing the deadline instruction semantics and integrating it into a thread-interleaved pipeline. We introduce a replaying mechanism to stall particular threads without stalling the entire pipeline. This replaying mechanism is again employed with the deadline instructions.

The Giotto language [14] is a novel approach to specifying system timing at the software level. However, it relies on the usual RTOS infrastructure that assumes worst-case execution time is known to establish schedulability [7]. Our PRET processor would be an ideal target for the Giotto programming environment; constructing one is future work.

Thread-interleaved pipelines date to at least 1987 [20], probably much earlier. Thread-interleaving reduces the area, power and complexity of a processor [11, 18], but more importantly, it promotes predictable execution of instructions in the pipeline. Access to main memory in thread-interleaved pipelines is usually pipelined [11, 18], but modern, large memories usually are not, so instead our approach presents each thread with a window in which it can access main memory. This provides predictable access to memory and mutual exclusion between the threads. We call this a memory wheel.

The goal of the Virtual Simple Architecture of Mueller et al. [2] is to enable hard real-time operation of unpredictable processors. They run real-time tasks on a fast, unpredictable processor and a slower, more-predictable one simultaneously, and switch over if the slow ever overtakes the fast. The advantage is that the faster processor will have time to run additional, non-time-critical tasks. By contrast, our PRET approach guarantees detailed timing, not just task completion times, allowing timing to be used for synchronization.

Scratchpad memories have long been proposed for embedded

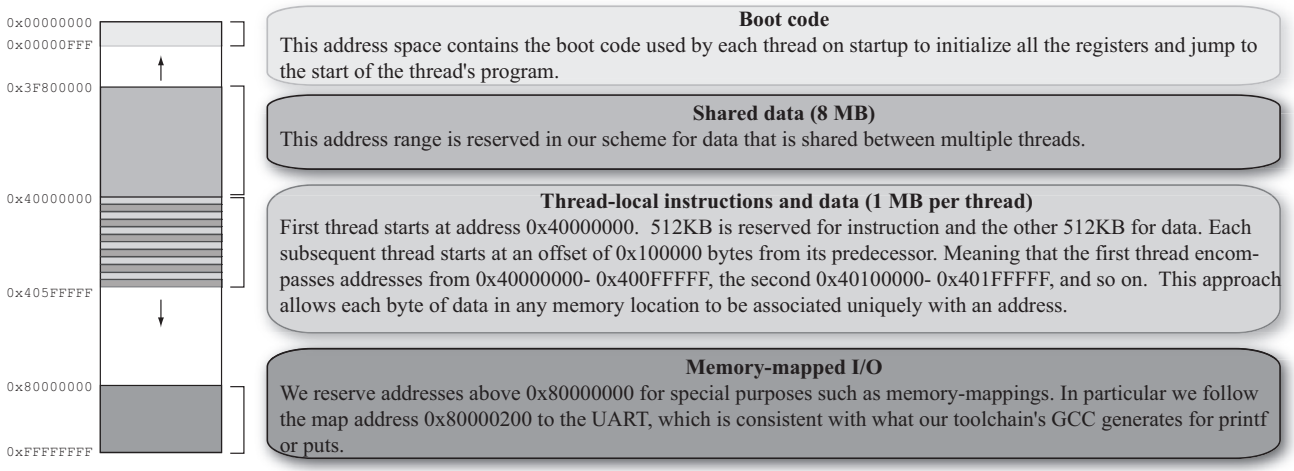


Figure 2: Memory Map

systems because they consume less power than caches [4], but here we adopt them purely because they enable better predictability. Since scratchpad memories are software managed, the issue of memory allocation schemes become important. Our future work is to build on top of the current PRET architecture and develop a memory allocation scheme.

4. OUR ARCHITECTURE

In this section, we present the design of the PRET processor, its memory system, and ISA extensions to support deadline counters. We have prototyped the PRET architecture (block diagram shown in Figure 1) with a cycle-accurate SystemC [23] model that executes programs written in C and compiled with the GNU C compiler. Our simulator implements an extended SPARC v8 ISA [28].

The `PRET PROCESSOR` component in Figure 1 implements a six-stage thread-interleaved pipeline in which each stage executes a separate hardware thread to avoid the need for bypasses [20, 11]. Each hardware thread has its own register file, local on-chip memory, and assigned region of off-chip memory. The `THREAD CONTROLLER` component is a simple round-robin thread scheduler—at any time, each thread occupies exactly one pipeline stage. To handle the stalling of the pipeline predictably, we introduce a replay mechanism that simply repeats the same instruction until the operation completes. Thus, the stalling of one thread does not affect any of the others. The round-robin execution of threads avoids memory consistency issues.

The memory hierarchy follows a Harvard architecture [13] that consists of separate fast on-chip scratchpad memories (SPM) for instruction and data, and a large off-chip main memory. They are connected to a direct memory access (DMA) controller responsible for moving data between main memory and the SPMs. Currently, we assume program code fits entirely in the SPMs because we have not developed an automatic program memory management scheme [3]. The DMA component currently only transfers the program code and data for each thread from main memory to their respective SPMs at power on. As mentioned earlier, memory reads and writes employ the replay mechanism. Each thread has its own access window managed by the `MEMORY WHEEL`. If a thread misses its window, it blocks until it reaches the start of its window.

We incorporate a deadline instruction [15] into our SPARC-based ISA. Such an instruction blocks until a software-programmable deadline counter reaches zero. Each counter is controlled by a thread-local phase-locked loop, which can be programmed to count at a rational multiple of the system clock frequency. Below, we describe our implementation in detail. We present the memory system, the memory wheel, the memory map, the thread interleaved pipeline, extension of the timing instructions and the toolchain flow.

4.1 Memory System

Caches are known to be a major source of timing unpredictability [30], but simply removing them is unacceptable. Instead, we use scratchpad memories for bridging the processor-memory gap. Scratchpad memories are software-managed on-chip memories often employed in hard real-time embedded processors. SPMs are managed by software through DMA transfers, thus avoiding the unpredictability of (often subtle) hardware replacement policies. Each thread-local SPM is 64 KB with 1 cycle latency. We use a 16 MB main memory which we assume to have a realistic latency of 50 ns. This translates to 12.5 cycles on a PRET processor running at approximately 250 MHz, and we round this up to 13 cycles.

If all threads were able to access the off-chip main memory at arbitrary times, then the off-chip memory access time for one thread could depend on the memory access pattern of another. This type of behavior introduces timing unpredictability and is undesirable. In order to ensure predictable timing, all reads and writes to main memory, such as shared data, must do so through our memory wheel. Like the "hub" in the Parallax Propeller Chip [24], this wheel has a fixed round robin schedule for determining which thread is allowed to access memory. Based on the fixed schedule and the time that a thread requests access to main memory, the access can take between 13 and 90 cycles. It is important to note that the exact number of cycles depends only on the cycle in which the request is made, and not on the behavior of other threads or memory access patterns.

Instead of blocking the entire pipeline during a multi-cycle memory access, we use the replay mechanism as described in the pipeline section. A simple memory management unit selects among the SPMs, the main memory, and memory-mapped I/O based on the address.

4.1.1 Memory Wheel

The memory wheel controls access to the off-chip main memory in a deterministic and predictable fashion. Each thread is allocated

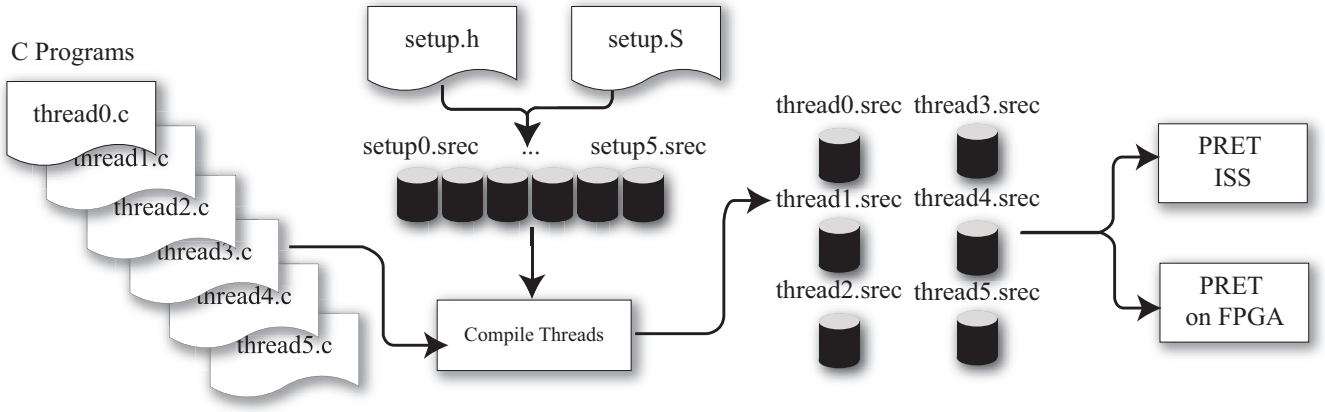


Figure 3: Compilation Flow for PRET Simulator

a 13-cycle window in which it must complete its memory access; the wheel schedule repeats every 78 cycles. If a thread starts its access on the first cycle of its window, the access takes exactly 13 cycles. Otherwise, the thread blocks until its window reappears, which may take up to 77 cycles. A successful access after just missing the first cycle of its window results in 77 + 13 = 90 cycles. While this mechanism can cause a thread to block, there is no inter-thread interaction and the behavior of the window is predictable.

4.1.2 Memory Map

Figure 2 shows the system memory map (addresses are 32 bits). Each piece of memory in the system has a unique global address (main memory and SPMs), but each thread only has access to part of the overall memory map. Addresses `0x3F800000` – `0x405FFFFF` (14 MB) are main memory, visible to every thread. This layout allows for future expansion: the shared data space can extend downward; thread-local memory can extend upward. Our current implementation has 512 bytes of boot code, 8 MB of shared data and 1 MB total for SPMs. Peripherals start at `0x80000000`; we placed a UART at `0x80000200`.

4.2 Pipeline

Our architecture implements a six stage thread-interleaved pipeline that supports six independent hardware threads. Each thread has its own set of registers. By design, we implemented one thread per pipeline stage to eliminate dependencies among instructions in the same thread. Thus, it does not need any data hazard detection logic or bypassing.

The thread controller schedules the threads according to a round-robin policy and passes a thread identifier to the pipeline that is used to index thread-specific registers and SPMs. In the `fetch` stage, the instruction pointed to by the current program counter is fetched from the SPM. Because of the thread interleaving, we do not update the program counter of the current thread. Instead, we update the program counter in the `except` stage. This is acceptable because the current thread will not be fetched until it exits the pipeline at the `except` stage. This removes the need for any speculative execution because we are sure the program counter is always correct when it is fetched. The `decode` stage decodes the instruction and sets the corresponding pipeline controls. The `regacc` stage reads the source operands from the register file and selects between immediate and registered data for the operands. The `execute` stage performs ALU operations. The `mem` stage accesses either SPM or main memory. The `except` stage catches exceptions and writes any results to the register file if there are no exceptions. We update the program counter in this stage after determining its next value depending on stalls and branches.

Even though we do not need data hazard detection, we do have to consider structural hazards. We address them with a replay mechanism, which we elaborate below.

4.2.1 Stalls and the Replay Mechanism

Our pipeline only updates registers in the `except` stage. This stage writes data to the register file only if no exception has occurred. This gives a single commit point for each thread. We decide at this commit point whether an instruction needs to be replayed.

By replaying instructions, we ensure each thread stays in exactly one pipeline stage per cycle before advancing, even for multi-cycle operations such as memory accesses. On a memory stall, for example, the stalled instruction is repeatedly replayed every six cycles until the data is fetched from memory and the thread can continue. Replay thus provides a predictable method for blocking a thread independently of the others, rather than stalling the whole pipeline. The exception stage checks an instruction's replay bit and commits only if the bit is clear. Otherwise, the `fetch` stage checks and determines the next program counter to be fetched. Therefore, the next iteration of that thread will re-run the same instruction until the multi-cycle operation is complete.

4.3 Timing Instructions

To provide precise timing control to software, we add a "deadline" instruction that allows the programmer to set and access cycle-accurate timers [15]. This instruction sets a lower bound deadline on the execution time of a segment of code. We provide two types of deadline timers that can be accessed by this instruction: one group counts according to the main clock, the other counts according to a clock generated by a programmable phase-locked loop (PLL). These timers appear as additional registers (Figure 1) that can only be accessed through the deadline instruction.

4.3.1 Syntax

We take the syntax of the deadline instruction from Ip and Edwards [15]. There is an immediate form, `deadi` $\$t_i, v$, and a register form, `dead` $\$t_i, \$r_j$. Each thread has twelve deadline registers (t_0–t_{11}), eight of which count instruction cycles, the other four are driven by the PLL; and 32 global registers (r_0–r_{31}). v is a 13-bit immediate value.

Producer

```
int main() {
  DEAD(28);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *buf = i;
  }
  return 0;
}
```

Consumer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  unsigned int i = 0;
  int arr[8];
  for (i =0; i<8; i++)
    arr[i] = 0;
  for (i = 0; ; i++) {
    DEAD(26);
    register int tmp = *buf;
    arr[i%8] = tmp;
  }
  return 0;
}
```

Observer

```
int main() {
  DEAD(41);
  volatile unsigned int * buf =
      (unsigned int*)(0x3F800200);
  volatile unsigned int * fd =
      (unsigned int*)(0x80000600);
  unsigned int i = 0;
  for (i = 0; ; i++ ) {
    DEAD(26);
    *fd = *buf;
  }
  return 0;
}
```

Figure 4: Simple Producer/Consumer Example

4.3.2 *Semantics*

The deadline instruction can only enforce a lower bound on the execution time of code segment; using replay, the deadline instruction blocks the thread whenever the deadline register t_i being written is not yet zero.

Unlike Ip and Edwards [15], our processor is pipelined, so we decrease each deadline register once every six clock cycles, i.e., at the instruction execution rate, and the PLL registers at the rate set by the PLL. When a deadline instruction attempts to set a deadline register, it blocks until the deadline register reaches zero, at which point it reloads the register and passes control to the next instruction. Thus, an earlier deadline instruction can set the minimum amount of time that can elapse before the next deadline instruction terminates.

Currently, if a deadline expires (i.e., the register reaches zero before a deadline instruction reaches it), we do nothing: the deadline instruction simply loads the new value immediately and continues. Later, we plan to allow the architecture to throw an exception when a deadline is missed.

4.3.3 *Implementation*

To implement the deadline instruction, we chose an unused opcode and followed the usual SPARC instruction coding format, which allowed us to include both register and immediate forms of the instruction. Figure 5 shows two concrete encodings.

Support for the deadline instruction requires some extra pipeline control logic and deadline registers.

11	00010	101100	00000	1	011111111111
op	rd	op3	rs1	i	simm13

11	00010	101100	00000	0	xxxxxxxx	00001
op	rd	op3	rs1	i	asi	rs2

Figure 5: Encoding of `dead $t2,0xFF` and `dead $t2,$g1`

In our pipeline, we check the deadline register in the register access stage and use the replay mechanism to block a deadline instruction until its deadline register is zero.

4.4 Compilation Flow

We adapted the SPARC toolchain used by the open-source LEON3 implementation [12]. Figure 3 shows our compilation flow.

We require the user to provide a `main()` function for each hardware thread in separate files (e.g., `thread0.c`). We compile each at locations dictated by our memory map by passing the `-Ttext` and `-Tdata` options to the linker. For example, `thread0.c` starts at address `0x40000000` and `thread1.c` at `0x40010000`. We merge the resulting object files with the setup code and convert them to Motorola S-record (SREC) format. Our simulator then initializes memory with the contents of the SREC files. We plan to use the same SREC files as input to our FPGA implementation of the PRET processor.

5. BASIC PRET PROGRAMMING

To illustrate how PRET timing precision can be used for synchronization, we present a simple producer/consumer example with an observer that displays the transferred data. This is a classical mutual exclusion problem in that we must deal with the issue of shared resources. Unlike the classical approach, however, the time that a thread must wait for a lock in our approach is deterministic in that it does not depend on the behavior of the other threads accessing the lock.

Our approach uses deadline counters and precise knowledge of the timing of instructions to synchronize access to a shared variable used for communication. We take on the role of a worst-case execution time (WCET) analysis tool to analyze the instructions generated from the C programs and compute the exact values for the deadline counters to ensure correct synchronization.

5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 4 shows the C code for the producer, consumer, and an observer all accessing the shared variable `buf` (underlined). The producer iterates and writes an integer value to the shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data. It just stores the data in the array. The observer also reads the shared data and writes it to a memory-mapped peripheral.

The deadline instructions in Figure 4 are marked in bold. We use staggered deadlines at the beginning of each thread to offset

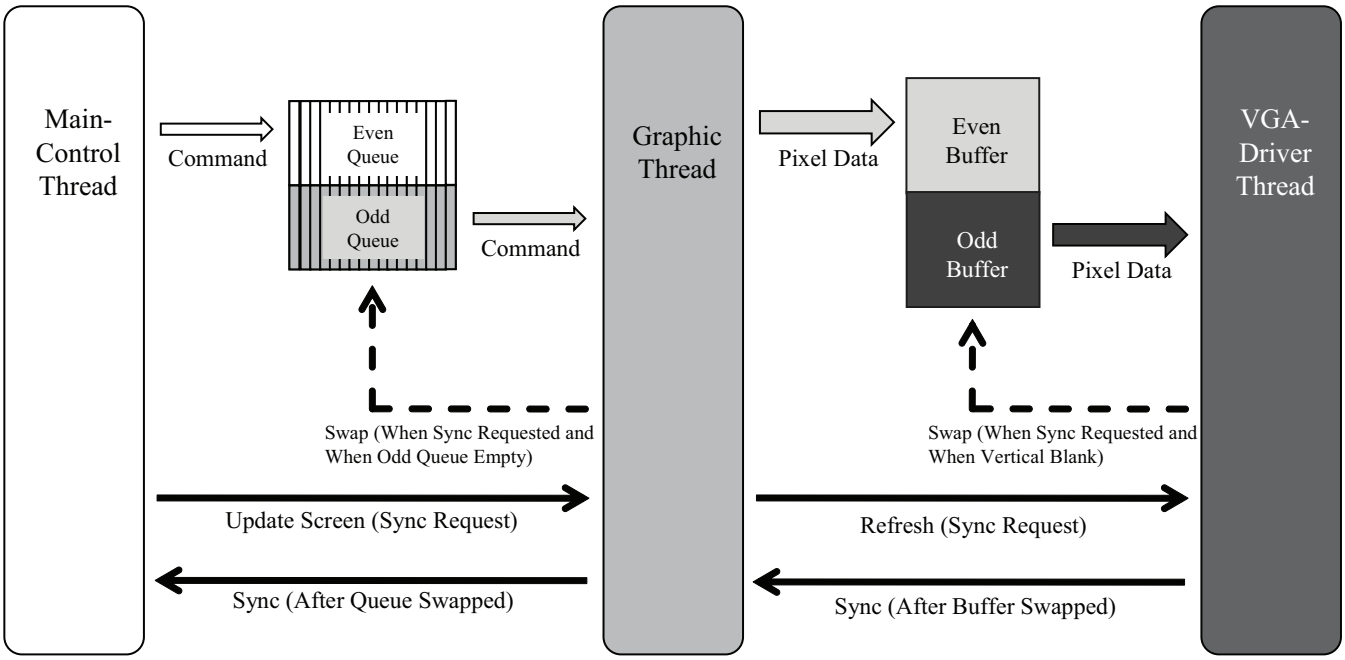


Figure 6: Structure of the Video Game Example

the threads and force the producer to start before the consumer and observer. Inside each thread, deadlines force each loop to run in lock-step with one another, each thread progressing at the same rate. Every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel.

The offsets to achieve this are given by deadlines at the beginning of the program. The deadline of 41, or $41 * 6 = 246$ cycles, is the same for the consumer and observer, and this forces both of these threads to enter the loop at the same time. This value is computed from the assembly language instructions and is the minimum deadline that the consumer thread can make. The offset of the producer loop is $28 * 6 = 168$ cycles, which is 78 cycles less than the offset of 246 for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the shared data during an earlier rotation of the wheel.

Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, we would like each loop iteration to take two rotations of the wheel. This allows us to write to the buffer in the first rotation of the wheel, and perform the reads for both the consumer and the observer in the second rotation of the wheel. This also means that our program will function correctly regardless of which C threads are partitioned to which hardware threads. To achieve this, we have set the deadlines in each of the loops to be 26, as $26 * 6 = 156$ cycles corresponds exactly two rotations of the wheel.

6. A SAMPLE APPLICATION: A VIDEO GAME

Inspired by an example game supplied with the Hydra development board [19], we implemented a simple video game in C targeted to our PRET architecture. Our example centers on rendering graphics and is otherwise fairly simple. The objective of the game is to avoid moving obstacles coming at the player's ship. The game ends when the player's ship collides with an obstacle. It uses multiple threads and both types of deadlines to meet its real-time requirements.

Signal	Time period	Pixel periods
Vertical sync	64μs	1641
Vertical back-porch	1.02ms	26153
Drawing 480 lines	15.25ms	
Vertical front-porch	350μs	8974
Horizontal sync	3.77μs	96
Horizontal back-porch	1.89μs	48
Drawing 640 pixels	25.17μs	
Horizontal front-porch	0.94μs	32

Table 1: VGA Real-time Constraints

Our example consists of three main tasks (Figure 6) running in separate threads: the video driver, which sends pixels in a framebuffer to a VGA port; the graphics controller, which is responsible for depositing pixels in the framebuffer; and the game logic. To the PRET simulator, we added a memory-mapped I/O interface to a VGA controller. The simulator dumps the pixel stream sent to this interface as PBM files for debugging. User input is currently read from an array.

For safety, we use a double-buffered command queue for communication between the game logic and the graphics controller, and a double-buffered frame buffer to communicate between the graphics controller and the video driver. During each frame, the game logic puts drawing commands into one of the command queues and the graphics controller interprets those in the other queue and draws them into one of the framebuffers. At the same time, the video controller is displaying the contents of the other framebuffer. This avoids screen flicker and guarantees the contents of each displayed frame is deterministic.

6.1 The VGA Driver Thread

This thread sends groups of sixteen pixels to the VGA controller to display a raster. As mentioned above, we double-buffer the image to avoid glitches: the VGA driver thread takes pixels from one buffer and the graphics controller writes to the other. The timing requirements, listed in Table 1, must be met to produce a stable image.

Our VGA driver displays four colors (black, white, red, and green) at 640×480 resolution with a 60 Hz refresh rate. We set the PLL clock to the 25.175 MHz pixel rate and fill a 32-bit hardware shift register every sixteen clocks. The VGA hardware takes a new pair of bits from this register at the pixel clock rate and sends them to a video DAC connected to a display.

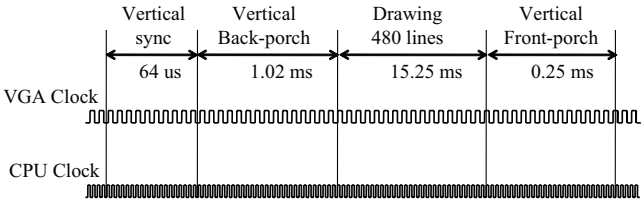


Figure 7: VGA Vertical Timing

The VGA driver algorithm is a simple loop: wait for the last pixels to be sent, read color or control data, send these to the VGA controller, and decide what to do next (typically repeat). In this context, control refers to horizontal and vertical synchronization signals. All of this happens at a rate dictated by the pixel-speed PLL clock.

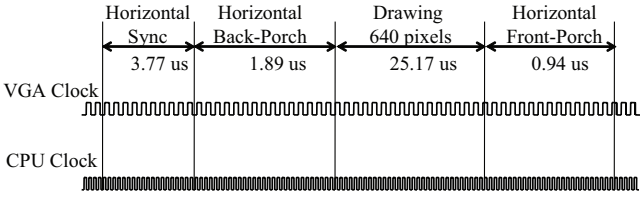


Figure 8: VGA Horizontal Timing

Synchronization requires careful timing. During vertical synchronization (Figure 7), vsync is asserted for 64 μs. The driver does this by using the PLL deadline instruction to wait 1641 pixel clocks after asserting vsync, then de-asserting it. We also use a deadline instruction to time the vertical backporch (1.02 ms) and the vertical frontporch (350 μs). The driver checks whether it should display the other buffer during vertical sync, and informs the graphics thread if it was requested.

Horizontal synchronization deadlines are more demanding. (Figure 8. Horizontal sync is asserted for 3.77 μs (96 pixel times), followed by a 1.89 μs frontporch (24 pixel times). Finally, we use the deadline instruction to control the speed at which data is fed to the video shift register. In a loop, we read data from the framebuffer, wait for the deadline to expire, then write the data to the shift register and update the fetch address.

Naturally, the inner pixel-drawing loop (Figure 9) is the most timing-critical. It requires six instructions: these six instructions must complete in under $16/25.175\text{MHz} = 635.6\text{ns}$. Five of the instructions take six cycles each. However, because it accesses main memory, the load instruction may take as many as 90 cycles, giving a total of $5 \times 6 + 90 = 120$ cycles overall. This requires a 5.3 ns clock period, or 188 MHz.

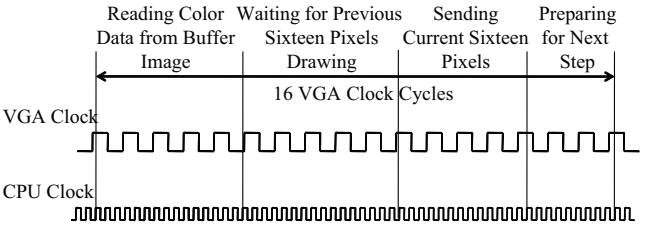


Figure 9: Timing of Sixteen Pixels

Pixel Drawing Routine

```
.LOOP:
   ld     [%o5], %g1     // load color data
   add    %o5, 4, %o5    // go to the next pixel
   deadi  %t9, 16        // wait for the deadline
   st     %g1, [%o3]     // write to the VGA shift register
   cmp    %o5, %o4       // done with the line?
   bne,a  .LOOP          // if not, repeat
```

Figure 10: Pixel Drawing Routine in Assembly

6.2 Graphics Thread

Following the example from the Hydra book, our graphics system is sprite-based: to draw to the framebuffer, the graphics thread assembles the overall image starting with a 640×480 pixel background, then stacks five 64×64 sprites on top of it. Each sprite may be placed at an arbitrary position on the screen. Figure 11 shows a typical image.

The graphics thread accepts three types of commands from the main thread through a double-buffered queue: drawing on the background or sprite layer, changing the position of the sprites, and filling the framebuffer according to the contents and position of the sprites and background image.

Ultimately, each displayed pixel is one of only four colors, but the pixels in our sprites can also be transparent. Such transparent pixels take on the color of any sprite beneath it or the background.

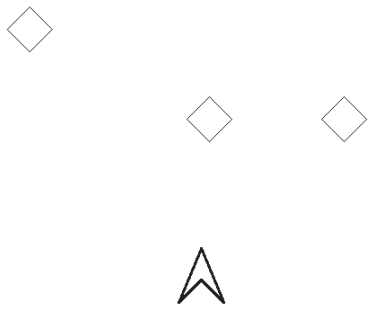
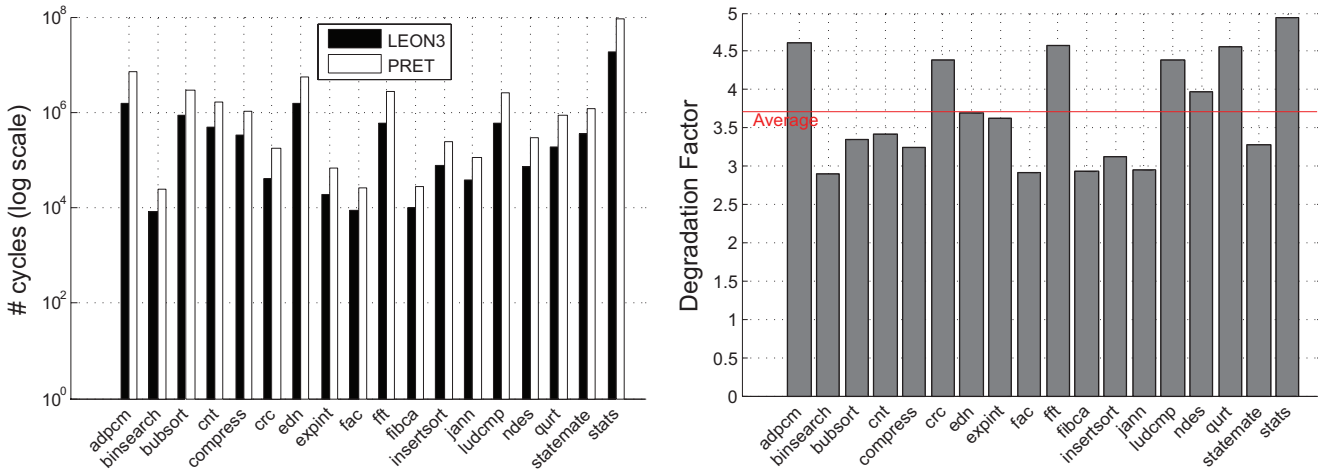


Figure 11: A Screen Dump From Our Video Game

6.3 Main Control/Game Logic Thread

The game logic thread takes user input, processes it, and sends commands to the graphic thread. At the moment, we take "user input" from an array; it should come from something like a joystick controller.



(a) Simulation Runs

(b) Degradation Factor

Figure 12: Simulation Runs for LEON3 and PRET Architectures

The game logic ends the commands for each frame with a screen update request. This prompts the graphics controller to redraw the framebuffer and also blocks the game logic thread until the redraw is complete. This synchronizes the game logic thread to the frame refresh rate, making it operate at a fixed rate.

6.4 Experience and Challenges with Programming for PRET

Since PRET's programming model has timing constructs, we were able to ensure the real-time constraints of the design with ease. The round-robin scheduling of the threads, predictable execution times of instructions and the deadline instructions make reasoning about the execution time of code segments straightforward.

However, the method to verify real-time requirement was error-prone; we calculated the timing constraints by hand. In this paper, we had to repeatedly calculate the timing requirement whenever our code is optimized or modified. In this way, it is hard to guarantee the calculated result is always right. Also, calculating by hand is slower than using automated tools for the calculation. We will provide automated methods for calculating and verifying timing constraints.

In addition, the lack of predictable synchronization primitives such as locks make inter-thread synchronization more challenging. This is because ISAs generally lack timing-predictable synchronization methods. In our game example, we synchronized the different threads by carefully investigating the timing and using deadline instructions as also shown by the simple producer/consumer example. The main advantage from this approach is that we can provide guarantees that the application meets its real-time requirements. We intend to introduce additional instructions that provide timing-predictable synchronization methods for easier programming of the PRET architecture.

7. SIMULATION RESULTS

Although we focus on programming for predictable timing, it is interesting to observe the average-case performance. We compare the performance of the PRET architecture against the LEON3 [12] SPARC processor. The LEON3 is an implementation of the SPARC v8 ISA for a system-on-chip design. It has a seven-staged conventional pipeline with instruction and data caches, and peripherals connected through an AMBA bus. We use a subset of the Malardalen WCET benchmarks [1] and simulate them on the LEON3 and PRET simulators. We compile the benchmarks using GCC's level 3 (`-O3`) optimization and software floating point (`-msoft-float`). For PRET, we load the benchmark on one hardware thread and leave the rest empty. Note that our hardware-multithreaded architecture pays a penalty for these completely sequential benchmarks, since five of its six hardware threads are left idle. One might expect that all of these benchmarks would require exactly six times as many cycles on our architecture as the single threaded SPARC processor, but due to the differences in instruction timings and caching behavior the difference is not that extreme. The cycle counts for the simulation runs are shown in Figure 12(a). We use a logarithmic scale for the cycle counts since benchmarks may execute for large varying number of instructions.

PRET shows a drop in average case performance with every benchmark. This degradation is expected because of PRET's thread-interleaved pipeline. Figure 12(b) shows the degradation factor for each of the benchmarks and the average degradation factor to be approximately 3.54 for the set of presented benchmarks.

Note that these benchmarks **do not** exercise the deadline instruction available in PRET. This is because most architecture such as the LEON3 do not support instructions with the deadline semantics. In addition, we map the benchmark to only one hardware thread. Thus, the simulation runs in Figure 12 do not make full use of PRET's multiple hardware threads.

8. FUTURE WORK

We have an enormous amount of future work planned for PRET. One obvious extension is to improve the scratchpad-to-main-memory link. Our current memory wheel, while a step in the right direction, is fairly naive. First of all, modern DRAM is usually banked and designed for burst transfers, yet we treat it as having uniform latency. It should be possible to give each thread a bank that it can access more quickly.

Modern large off-chip memories are set up for fast burst transfers designed to fill cache lines, yet our current architecture does not take advantage of this. To make the most of shared memory, it would be nice if the thread could move a block of memory to its scratchpad during each turn. The SPARC ISA does not support this at the moment; we plan to add it.

How software is written can greatly affect how efficiently the memory is used. We plan to integrate many of the algorithms that have been developed for managing scratchpad memory, both for code (traditionally thought of as overlays) and for data. Integrating these with the particular memory movement mechanisms we develop is one of our next projects.

The memory wheel is one example of what we will expect to be many time-triggered components in a PRET system. An obvious challenge is how to make best use of it by choosing to try to access it at the optimal time. We envision a compiler able to reason about when each instruction will execute (PRET, of course, makes this possible) and thus about when best to attempt access to, say, main memory. This will work something like Dean's software thread integration [31], in which a compiler mindful of instruction timing restructures the code to meet real-time deadlines. For periodic-access components in a PRET setting, we would probably color each instruction with its phase relative to when the thread could access main memory and attempt to reorder instructions to minimize waiting time.

A compiler for a PRET system would go one step beyond existing C compilers and perform WCET analysis as a normal part of its operation. Perhaps with some user annotation support, it should be able to determine things like loop bounds and see whether the code can meet every deadline. Our goal is to make a timing error as easy to detect, understand, and correct as a syntax error.

We presented a single-cored PRET machine in this paper, but we plan to extend PRET to multi-core configurations. Much of the basic architecture—the scratchpads and timers—will remain unchanged in this setting, but access to main memory and mechanisms for inter-core communication will have to be added. We envision continuing to take a time-triggered approach in which access to a shared resource like a communications network will be arbitrated periodically.

Methods for evaluating timing predictability are needed. Our performance results compared the LEON3 with the PRET machine in terms of cycles taken to execute the same set of instructions. This approach does not compare the timing predictability of the processors. For these reasons, we plan on defining a method for testing timing predictability in the future.

9. CONCLUSION

In this paper, we described an architecture that delivers predictable timing and implemented it as a cycle-accurate SystemC model that executes C programs. Our PRET architecture implements and extends the SPARC ISA with a precise timing control instruction called deadline. It presents a two-level memory hierarchy composed of thread-local scratchpad memories and a shared main memory accessed through a memory wheel. We illustrated the programming of a PRET architecture with a simple producer/consumer example and a larger video game example.

We compared the performance of the PRET core against the LEON3 embedded processor. Our results showed the PRET core to be slower. However, this is an expected degradation in performance caused by the thread-interleaved pipeline architecture of PRET. Note that our comparison favors the LEON3 processor over the PRET architecture. This is because we do not compare the timing predictability of the two processors, which is PRET's forte.

10. ACKNOWLEDGMENTS

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) and #0720841 (CSR-CPS)), 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, HSBC, Lockheed-Martin, National Instruments, and Toyota.

11. REFERENCES

[1] Malardalen WCET project / Benchmarks. Website: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[2] 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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 350–361, San Diego, California, June 2003.

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

[4] L. Benini, A. Macii, E. Macii, and M. Poncino. Increasing Energy Efficiency of Embedded systems by Application-Specific Memory Hierarchy Generation. *IEEE Design & Test of Computers*, 17(2):74–85, April-June 2000.

[5] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] H. Berry, D. G. Pérez, and O. Temam. Chaos in computer performance. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 16(013110):1–15, Jan. 2006.

[7] E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, 2004.

[8] M. Delvai, W. Huber, P. Puschner, and A. Steininger. Processor support for temporal predictability-the SPEAR design example. *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference*, pages 169–176, 2003.

[9] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proceedings of the 44th Design Automation Conference*, pages 264–265, San Diego, California, June 2007.

[10] C. Ferdinand, R. Heckmann, and et. al. Reliable and precise WCET determination for a real-life processor. In *Proceedings of the International Conference on Embedded Software (EMSOFT)*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485, North Lake Tahoe, California, Oct. 2001.

[11] B. Fort, D. Capalija, Z. Vranesic, and S. Brown. A Multithreaded Soft Processor for SoPC Area Reduction. *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'06)-Volume 00*, pages 131–142, 2006.

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

[13] J. L. Henessey and D. J. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, third edition, 2003.

[14] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.

[15] N. J. H. Ip and S. A. Edwards. A processor extension for cycle-accurate real-time software. In *Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 4096 of *Lecture Notes in Computer Science*, pages 449–458, Seoul, Korea, Aug. 2006.

[16] S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, 2004.

[17] H. Kopetz and G. Grünsteidl. TTP — A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1):14–23, Jan. 1994.

[18] M. Labrecque and J. G. Steffan. Improving pipelined soft processors with multithreading. In *7th International Conference on Field Programmable Logic and Applications (FPL)*, Amsterdam, Netherlands, August 2007.

[19] A. LaMothe. *Game Programming for the Propeller Powered HYDRA*. Parallax, Inc., Rocklin, California, 2006.

[20] E. A. Lee and D. G. Messerschmitt. Pipeline interleaved programmable DSP's: Architecture. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, ASSP-35(9):1320–1333, Sept. 1987.

[21] X. Li, J. Lukoschus, M. Boldt, M. Harder, and R. von Hanxleden. An Esterel processor with full preemption support and its worst case reaction time analysis. *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 225–236, 2005.

[22] J. Miller and A. Agarwal. Software-based instruction caching for embedded processors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 293–302, San Jose, California, Oct. 2006.

[23] Open SystemC Initiative (OSCI). SystemC Simulation Library. Website: http://www.systemc.org.

[24] Parallax Inc. Propeller Manual. Website: http://www.parallax.com/Portals/0/Downloads/docs/prod/prop/WebPM-v1.01.pdf.

[25] D. A. Patterson and D. R. Ditzel. The case for the reduced instruction set computer. *ACM SIGARCH Computer Architecture News*, 8(6):25–33, Oct. 1980.

[26] Z. Salcic, D. Hui, P. Roop, and M. Biglari-Abhari. REMIC: design of a reactive embedded microprocessor core. *Proceedings of the 2005 conference on Asia South Pacific design automation*, pages 977–981, 2005.

[27] M. Schoeberl. JOP: A Java Optimized Processor. *Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2003), Catania, Sicily, Italy, November*, 2003.

[28] SPARC International Inc. SPARC Standards. Website: http://www.sparc.org.

[29] M. B. Taylor, J. Kim, and et al. The Raw microprocessor: A computational fabric for software circuits and general purpose programs. *IEEE Micro*, 22(2):25–35, March/April 2002.

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

[31] B. J. Welch, S. O. Kanaujia, A. Seetharam, D. Thirumalai, and A. G. Dean. Supporting demanding hard-real-time systems with STI. *IEEE Transactions on Computers*, 54(10):1188–1202, Oct. 2005.