Predictable Programming on a Precision Timed Architecture

Ben Lickly
Isaac Liu
Sungjun Kim
Hiren D. Patel
Stephen A. Edwards
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-40

http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-40.html

April 18, 2008

Copyright © 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Edwards is supported by NSF, Intel, Altera, the SRC, and NYSTAR. Lee received support for this work 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), #0647591 (CSR-SGER), 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 and AF-TRUST #FA9550-06-1-0244), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, DGIST, National Instruments, and Toyota.

Predictable Programming on a Precision Timed Architecture

Ben Lickly†, Isaac Liu†, Sungjun Kim°,
Hiren D. Patel†, Stephen A. Edwards° and Edward A. Lee†

† Electrical Engineering and Computer Science
University of California, Berkeley,
Berkeley, CA 94720, USA

° Department of Computer Science
Columbia University,
NY 10027, USA

April 18, 2008

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.

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 [8], let alone code with conditional branches. Modern processors are truly chaotic [4].

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 [7], 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 [21] 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 [7] 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 [23] started the RISC revolution. In the same way, we must rethink real-time embedded processor architectures.

The complexity of modern processors [11] has made the task of calculating or even bounding the execution time of a sequence of operations very difficult [8]. 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 [2], thread-interleaved pipelines with no bypassing [9, 18], explicit timing control at the ISA level [13], time-triggered communication [15] with global time synchronization [14], and high-level languages with explicit timing [12]. 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. [27] shares certain elements of the PRET philosophy. It, too, employs a software-managed scratchpad instead of an instruction cache [20], 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 [25] enables accurate worst-case execution time bounds, but does not provide support for controlling execution time. The SPEAR [6] processor prohibits conditional branches, which we find overly restrictive. The REMIC [24] and KIEL [19] are predictable in the PRET sense, but they only allow Esterel 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 [13] 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.

The Giotto language [12] 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 [5]. 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 [18], probably much earlier. Thread-interleaving reduces the area, power and complexity of a processor [9, 16], but more importantly, it promotes predictable execution of instructions in the pipeline. Access to main memory in thread-interleaved pipelines is usually pipelined [9, 16], 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. [1] 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 systems because they consume less power than caches [3], 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 [21] model that executes programs written in C and compiled with the GNU C compiler. Our simulator implements an extended SPARC v8 ISA [26].

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 [9,18]. 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

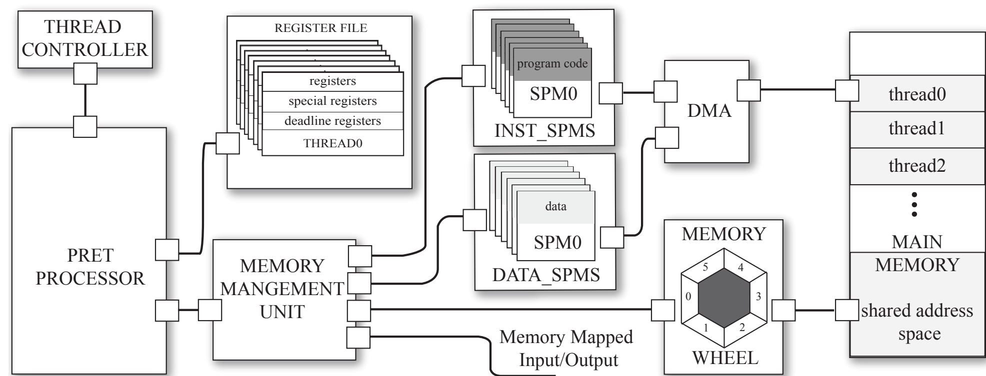

Figure 1: Block Diagram of PRET Architecture

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 [11] 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 [2]. 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 [13] 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 [28], 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 with a latency of 13 cycles. All reads and writes to the main memory, such as shared data, must do so through our memory wheel. Like the "hub" in the Parallax Propeller Chip [22], 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. 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 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. 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`.

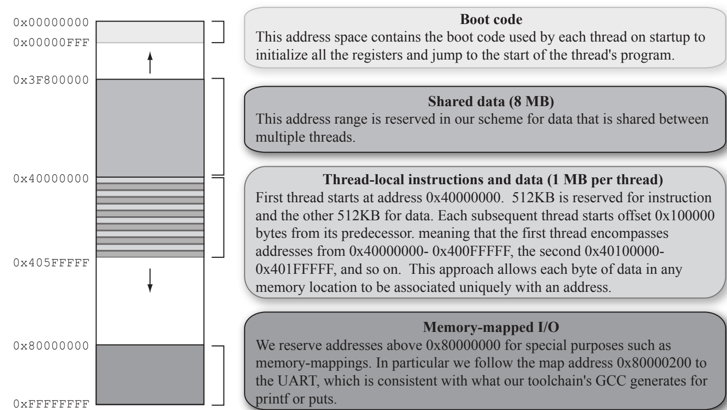

Figure 2: Memory Map

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 to index thread-specific registers and SPMs. In the `fetch` stage, the current PC is fetched from the SPM. Because of the thread interleaving, we do not need to update the next PC for the current thread since it will not be fetched until the current thread leaves the pipeline. This removes the need for something like speculative execution because we are sure the PC 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 also update the PC 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. 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 PC 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 [13]. 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 [13]. 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.

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 [13], 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 continue. 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 3 shows two concrete encodings.

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

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.

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

4.4 Compilation Flow

We adapted the SPARC toolchain used by the open-source LEON3 implementation [10]. Figure 4 shows our compilation flow.

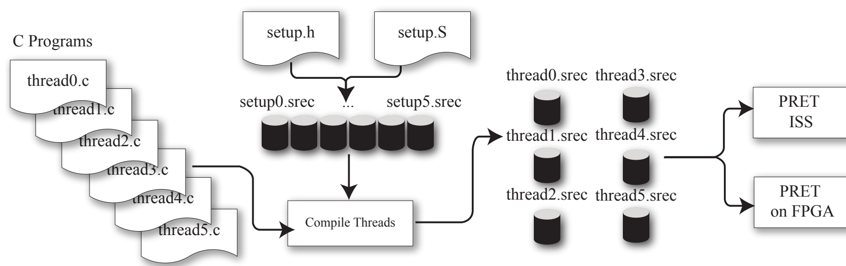

Figure 4: Compilation Flow for PRET Simulator

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 setup code and convert them to Motorola S-record (SREC) format. Our simulator then initializes memory with the contents of the SREC files.

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; we must deal with the issue of shared resources. We address this using the 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.

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 5: Simple Producer/Consumer Example

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 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a 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 or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, 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 offset of the producer loop is $28*6 = 168$ cycles, which is 78 cycles less than the offset of $41*6 = 246$ 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 data 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, each loop iteration takes $26*6 = 156$ cycles: exactly two rotations of the wheel.

6 A Sample Application: A Video Game

Inspired by an example game supplied with the Hydra development board [17], 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.

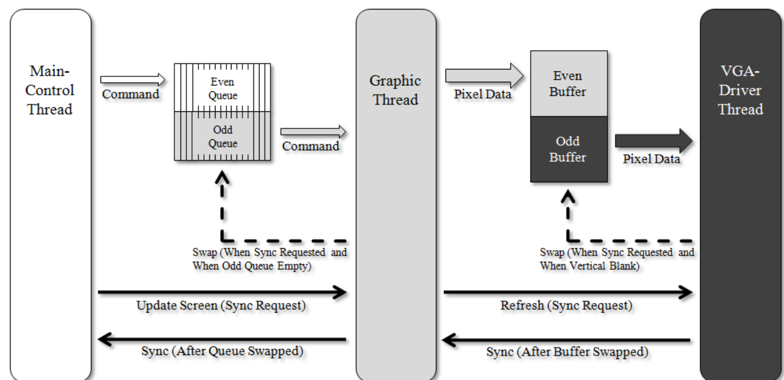

Figure 6: Structure of the Video Game Example

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.

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

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.

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.

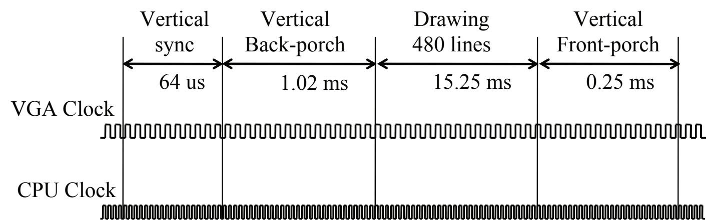

Figure 7: VGA Vertical Timing

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.

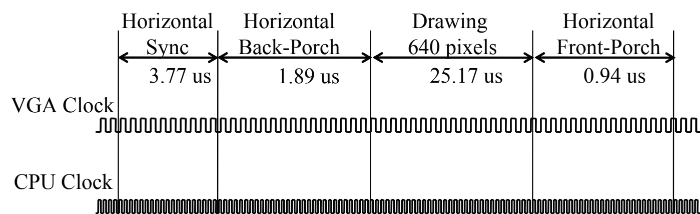

Figure 8: VGA Horizontal Timing

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.

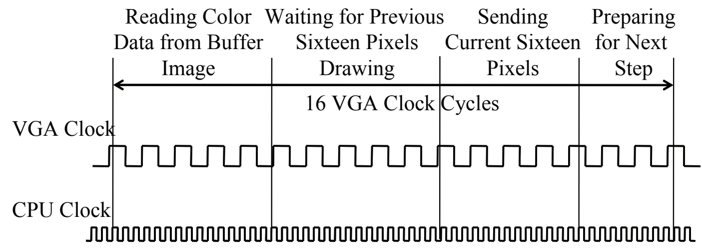

Figure 9: Timing of Sixteen Pixels

Naturally, the inner pixel-drawing loop (Figure 9) is the most timing-critical. It requires six instructions:

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
```

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.

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 10 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.

6.3 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.

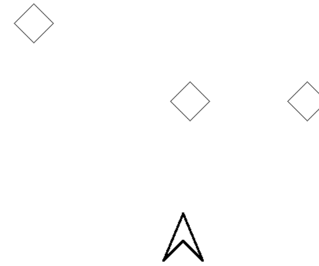

Figure 10: A Screen Dump From Our Video Game

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-robing 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 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 Conclusions

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 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 [29], 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.

References

[1] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, Eric Rotenberg, and Frank 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.

[2] Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. Embedded Computing Sys.*, 1(1):6–26, 2002.

[3] Luca Benini, Alberto Macii, Enrico Macii, and Massimo 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.

[4] Hugues Berry, Daniel Gracia Pérez, and Olivier Temam. Chaos in computer performance. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 16(013110):1–15, January 2006.

[5] Enrico Bini and Giorgio C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Trans. Computers*, 53(11):1462–1473, 2004.

[6] 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 on*, pages 169–176, 2003.

[7] Stephen A. Edwards and Edward 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.

[8] Christian Ferdinand, Reinhold 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, October 2001.

[9] B. Fort, D. Capalija, Z.G. Vranesic, and S.D. 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.

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

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

[12] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A time-triggered language for embedded programming. In *EMSOFT 2001*, volume LNCS 2211, Tahoe City, CA, 2001. Springer-Verlag.

[13] Nicholas Jun Hao Ip and Stephen 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, August 2006.

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

[15] Hermann Kopetz and Günter Grünsteidl. TTP — a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

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

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

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

[19] 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.

[20] Jason Miller and Anant 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, October 2006.

[21] OSCI. SystemC Simulation Library. Website: http://www.systemc.org.

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

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

[24] 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.

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

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

[27] Michael Bedford Taylor, Jason 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.

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

[29] Benjamin J. Welch, Shobhit O. Kanaujia, Adarsh Seetharam, Deepaksrivats Thirumalai, and Alexander G. Dean. Supporting demanding hard-real-time systems with STI. *IEEE Transactions on Computers*, 54(10):1188–1202, October 2005.