Preliminary work as a Poster

Isaac Liu, Ben Lickly, Hiren D. Patel, and Edward A. Lee, "Poster Abstract: Timing Instructions - ISA Extensions for Timing Guarantees," a poster at Real-Time and Embedded Technology and Applications Symposium (RTAS), San Francisco, California, 2009.

Poster Abstract: Timing Instructions– ISA Extensions for Timing Guarantees

Isaac Liu, Ben Lickly, Hiren D. Patel, and Edward A. Lee
Center for Hybrid and Embedded Software Systems, EECS,
University of California, Berkeley,
Berkeley, CA 94720, USA
{liuisaac, blickly, hiren, eal}@eecs.berkeley.edu

***Abstract*—We present our on-going efforts to guarantee the timing behavior of a program targeted for the precision timed architecture. We extend both the ISA and the hardware to support a set of timing instructions that allow programmers to control the execution time of a sequence of instructions. Programs written using these timing instructions specify deadlines within the program specification itself, and the hardware architecture enforces them through specific hardware policies. For example, timing instructions may be used to ensure that a segment of code has repeatable timing behavior, or that when a timing requirement is violated, an exception handler is invoked to appropriately address the timing violation. In this paper, we present the supported timing instructions, their semantics, and illustrative examples of their usage.**

I. Introduction

Instruction-set architectures (ISAs) serve as contracts between the software and the hardware. Modern computer architectures that implement these ISAs are free to use microarchitectural techniques that enhance the average-case performance. While these ISAs and their high-performance implementations are suitable for writing general purpose software, we find them to be problematic for real-time embedded software. This is because timeliness in real-time embedded software is just as important as correct functionality. Since ISAs do not provide any means of exposing or controlling the timing behavior of software, their implementations are under no obligations to exhibit predictable and repeatable timing behaviors. As a result, predicting execution times and getting repeatable timing behaviors from software on modern computer architectures [1] is virtually impossible.

We believe the solution for real-time embedded software is a rethinking of both the ISA, and the processor architecture. As has been argued in [2], we must consider architectures that provide timing as predictable and repeatable as their function. Our initial prototype is the precision timed (PRET) architecture [3], [4]. It is a real-time embedded processor that judiciously selects architectural optimization techniques to deliver performance without sacrificing timing predictability and repeatability. In addition to the processor architecture, we make time a fundamental property of the ISA by extending it with timing instructions. These timing instructions bring timing controllability up to the abstraction level of the software, by allowing control over the timing behavior of a piece of code. We can use timing instructions to enforce certain timing requirements, and also handle timing-exceptions when the timing requirements are violated. In this paper, we present an overview of the PRET architecture, and our current efforts in extending the ISA with a series of timing instructions. We describe their semantics, and through illustrative examples we describe their usage.

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, Lockheed-Martin, National Instruments, and Toyota

II. PRET Architecture

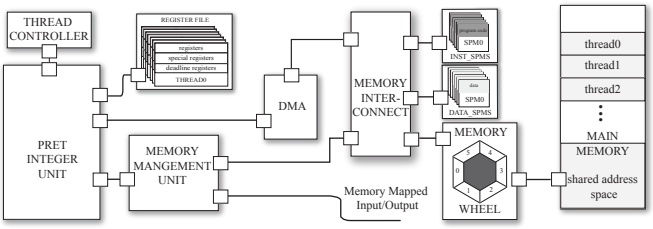

Fig. 1. Block Diagram of PRET Architecture

Figure 1 shows the a block diagram of the PRET architecture. The `INTEGER UNIT` contains the main pipeline of the architecture. Conventional pipelines use speculative techniques such as branch predictions to mitigate the performance penalty from hazards. These give better average-case performance, but consequently make the pipeline even more unpredictable. Our architecture uses a thread-interleaved pipeline to remove hazards [5]. A thread-interleaved pipeline cycles hardware threads in a round-robin fashion each cycle for execution in the pipeline. This removes the data and control hazards, which allows us to omit the branch predictor as well as the typical pipeline forwarding and bypassing logic to gain predictability.

We use a Harvard architecture for our memory, but instead of using caches, which result in unpredictable execution time, we employ scratchpad memories [6]. This brings the control of fast access memory to software, and avoids policies implemented in hardware. This gives us better execution time estimates at compile time. In order to decouple the access

time to memory of different hardware threads, all memory access goes through a `MEMORY WHEEL` that grants hardware threads time-triggered access to the main memory. This way, we have a strict upper bound on the time to access main memory independently of the memory access patterns for each thread. Instead of stalling the whole pipeline, we use a replay mechanism to replay instructions that are waiting for memory or another operation to complete. This further decouples the execution time of different hardware threads, and it allows us to predictably analyze the execution time of each thread without worrying about interference from other hardware threads.

We have prototyped the PRET architecture as a cycle-accurate SystemC model that executes programs written in C, and compiled with the GNU GCC or LLVM C compilers. Our simulator implements a SPARC v8 ISA [7] with timing instruction extensions. We are currently implementing our prototype as a softcore.

III. Timing Instructions

In order to expose the timing control to software, we have augmented the traditional SPARC ISA with "deadline" instructions in the fashion of Ip and Edwards [8], which allow the programmer to set and access cycle-accurate timers. Our set of deadline instructions are extended, however, allowing a broader range of time-dependent behaviors to be encoded. These instructions offer precise timing control of the code that they enclose (the *deadline block*), by explicitly specifying the number of clock cycles that should pass before the next instruction is completed. If the enclosed code executes before its timer has expired, it may be deterministically stalled using the replay mechanism to run at the correct rate. If the provided deadline runs out before the enclosed code has finished, it can throw an exception and run the appropriate handling code.

A table showing the various types of instructions supported is given in Table I. In general, the timing instructions work on a special set of registers that are decremented every cycle. The load instructions can load the initial deadline value into a register and start a deadline block.

TABLE I
List of deadline instructions added to ISA

Timing instruction	Behavior
`deadload`	Load a new deadline value into the given timer register.
`dead`	Stall until previous timer expires, then load new value into timer.
`deadbranch`	Same as `dead`, but raise an exception if new deadline expires.
`deadloadbranch`	Combination of functionality of `deadload` and `deadbranch`.

If the following timing instruction is not a load-type instruction, then it stalls execution until the timer reaches zero. We say that this block of code has *met* its deadline. This is desirable because it enforces a repeatable execution time for the enclosed code.

In cases where we cannot guarantee that the block will meet its deadline, we can make the first timing instruction a branch-type instruction. In this case, if the deadline for the block is not met, rather than simply continuing, an exception will be raised. The program can also specify what action should be taken at runtime to address the missed deadline.

Fig. 2. Examples of patterns that use timing instructions

Enforcing WCET

```
deadloadbranch %t1,0x1000
... ; enclosed code must
... ; complete within
... ; 0x1000 cycles
deadload %t1,0x0
```

(a) Limit the runtime of a code block

Fixed Period Loops

```
loop:
  dead %t1,0x700
  ... ; each iteration must
  ... ; take 0x700 cycles
  b loop
```

(b) Set the period of a loop

A. Examples

In Figure 2, we illustrate the usage of a few representative programming patterns that utilize timing instructions. In the first example, an upper bound on the execution time of a segment code is enforced with branch-type deadlines, which call special recovery code in case the enclosed code does not meet its deadline of 0x1000 cycles. In the second example, we show how to use a deadline timing instruction in a loop to ensure that the loop runs no faster than it is allowed. In both of these examples, the timing instructions explicitly specify a timing constraint in the code of the program.

IV. Future Work & Conclusion

As the size and complexity of real-time embedded systems continues to grow, it will become increasingly difficult to verify and guarantee the timing properties of the system without suitable abstractions. With timing predictability as its core design principle, PRET allows real-time embedded systems to be built upon a solid foundation with predictable temporal behaviors. We are looking to improve the programmability of the architecture, allowing simpler and more flexible use of the timing instructions. At the same time, we plan to provide compiler support for memory management and timing verification, as well as do more architectural exploration for precise handling of concurrent events.

References

[1] C. Ferdinand, R. Heckmann, and et. al, "Reliable and precise WCET determination for a real-life processor," vol. 2211, North Lake Tahoe, California, Oct. 2001, pp. 469–485.

[2] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," *Proceedings of the 44th Design Automation Conference*, pp. 264–265, June 2007.

[3] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee, "Predictable Programming on a Precision Timed Architecture." Proceedings of International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES), 2008, pp. 137–146.

[4] PRET Group, "Precision Timed Architecture Simulator v1.0," Website: http://www.chess.eecs.berkeley.edu/pret/release/.

[5] E. A. Lee and D. G. Messerschmitt, "Pipeline interleaved programmable DSP's: Architecture," vol. ASSP-35, no. 9, pp. 1320–1333, Sep. 1987.

[6] O. Avissar, R. Barua, and D. Stewart, "An optimal memory allocation scheme for scratchpad-based embedded systems," *ACM Transactions on Embedded Computing Systems*, vol. 1, no. 1, pp. 6–26, 2002.

[7] SPARC International Inc., "SPARC Standards," Website: http://www.sparc.org.

[8] N. J. H. Ip and S. A. Edwards, "A processor extension for cycle-accurate real-time software," Master's thesis, Columbia University.