Precision Timed (PRET) Computation in Cyber-Physical Systems

Edward A. Lee (UC Berkeley) and
Stephen A. Edwards (Columbia University) *

Updated Position Paper for the *National Workshop on High Confidence Software Platforms for Cyber-Physical Systems: Research Needs and Roadmap*
November 30 - December 1, 2006, Alexandria, Virginia

Date of Update: January 8, 2007

Cyber-Physical Systems (CPS) are integrations of computation with physical processes. Embedded computers and networks monitor and control the physical processes, usually with feedback loops where physical processes affect computations and vice versa. In the physical world, the passage of time is inexorable and concurrency is intrinsic. Neither of these properties is present in today's computing and networking abstractions. As a consequence, these abstractions require some fundamental rethinking.

It is tempting to believe that the CPS problems can be solved by overlaying higher-level abstractions on top of existing computing technology. Indeed, it would be a scary prospect to suggest that much of the foundation of existing technology is flawed and must be rebuilt. How could this possibly result in a practical research program that will see results in our lifetime?

In this position paper, we make a case that core computing abstractions must be and can be effectively and practically rebuilt. The objective is to enable a new generation of cyber-physical systems where computation and physical processes are tightly intertwined. This requires re-introducing properties that were deliberately and systematically abstracted away in the 20-th century view of computation. We approach the problem bottom-up. We must first rebuild the computational engines, and then build revised higher level abstractions on top of these.

1 The Problem

In 1980, Patterson and Ditzel [12] did not invent reduced instruction set computers (RISC). Earlier computers all had reduced instruction sets. Instead, they argued that trends in computer architecture had gotten off the sweet spot, and that by dropping back a few years and forking a new version of architectures, leveraging what had been learned, they could get better computers by employing simpler instruction sets.

It is again time for a change in direction in computer architecture. Architectures currently strive for superior average-case performance that regrettably ignores predictability and repeatability of timing properties. "Correct" execution of the SPECint benchmark suite has nothing to do with how long it takes to perform any particular action. C says nothing about timing, so timing is not considered part of correctness. Architectures have developed deep pipelines with speculative execution and dynamic dispatch. Memory architectures have developed multi-level caches and TLBs. The performance criterion is simple: faster (on average) is better.

The biggest consequences have been in embedded computing. Avionics offers an extreme example: in "fly by wire" aircraft, where software interprets pilot commands and transports them to actuators through networks, certification of the software is extremely expensive. Regrettably, it is not the software that is certified but the entire system. If a manufacturer expects to produce a plane for 50 years, it needs a 50-year stockpile of fly-by-wire components that are all made from the same mask set on the same production line. Even a slight change or "improvement" might affect timing and require the software to be re-certified. For good reason, the FAA does not trust software.

Figure 1 illustrates schematically some of the abstraction layers on which we depend when designing embedded systems. In this three-dimensional Venn diagram, each box represents a set. For example, at the bottom, we have the set of all microprocessors. An element of this set, e.g. the Intel P4-M 1.6GHz, is a particular microprocessor. Above that, we have the set of all x86 programs, each which can run on that microprocessor. This set is defined precisely (unlike the previous set, which is hard to define precisely) by the definition of the x86 instruction set architecture (ISA). Any program written using that instruction set is a member of the set. For example, a particular implementation a Java virtual machine is a member of the set. Associated with that member is another set, the set of all JVM byte-code programs. Each of these programs is (typically) synthesized by a com-

*Lee received support this work from NSF award number CNS-0647591. Edwards is supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

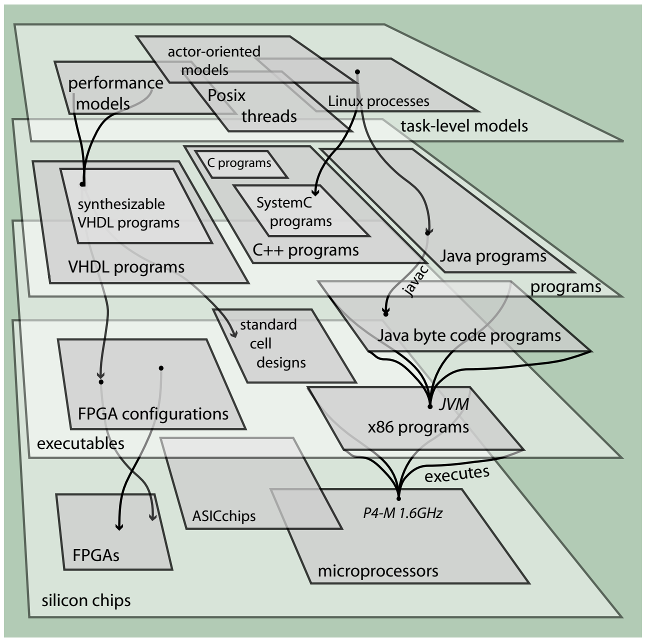

Figure 1: Abstraction layers in computing.

piler from a Java program, which is a member of the set of all syntactically valid Java programs. Again, this set is defined precisely by the Java syntax.

Each of these sets provides an abstraction layer that is intended to isolate a designer (the person or program that selects elements of the set) from the details below. Many of the best innovations in computing have come from careful and innovative construction and definition of these sets.

However, in the case of our poor aircraft manufacturer, nearly every abstraction has failed. The instruction-set architecture, meant to hide hardware implementation details from the software, has failed because the user of the ISA cares about timing properties that the ISA does not guarantee. The programming language, which hides details of the ISA from the program logic, has failed because no widely used programming language expresses timing properties. Timing is merely an accident of the implementation. A real-time operating system hides details of the programs from the concurrent orchestration, yet this fails because the timing may affect the orchestration. The RTOS provides no guarantees. The network hides details of electrical or optical signaling from systems, but standard networks provide no timing guarantees, and hence again fail to provide an appropriate abstraction. The aircraft manufacturer is stuck with a system *design* (not just implementation) in silicon and wires.

All embedded systems designers face less extreme versions of this problem. "Upgrading" a microprocessor in an engine control unit for a car requires thorough re-testing of the system. Even "bug fixes" in the software can be extremely risky, since they can change timing behavior and produce effects that were never seen in testing.

The design of an abstraction layer involves many choices, and computer scientists have chosen to hide timing properties of physical realizations from all higher abstractions. Wirth [13] says "It is prudent to extend the conceptual framework of sequential programming as little as possible and, in particular, to avoid the notion of execution time." In the context of embedded systems, however, computations interact directly with the physical world, where time cannot be abstracted away. But even general-purpose computing suffers from these choices. Since timing is neither specified in programs nor enforced by execution platforms, a program's timing properties are not repeatable. Buggy concurrent software often has timing-dependent behavior; small changes in the timing of one part of a program can affect seemingly unrelated parts.

Designers have traditionally covered these failures by finding worst case execution time (WCET) bounds and using real-time operating systems (RTOS's) with predictable scheduling policies. But these require substantial margins for reliability, and ultimately reliability is (weakly) determined by bench testing of the complete implementation. Moreover, WCET has become an increasingly problematic fiction as processor architectures develop ever more elaborate techniques for dealing stochastically with deep pipelines, memory hierarchy, and parallelism.

Modern processor architectures render WCET virtually unknowable; even simple problems demand heroic efforts. For example, Ferdinand et al. [5] determine the WCET of astonishingly simple avionics code from Airbus running on a Motorola ColdFire 5307, a pipelined CPU with a unified code and data cache. Despite the software consisting of a fixed set of non-interacting tasks containing only simple control structures, their solution requires detailed modeling of the seven-stage pipeline and its precise interaction with the cache, generating a large integer linear programming problem. The technique successfully computes WCET, but only with many caveats that are increasingly rare in software. Fundamentally, the ISA of the processor has failed to provide an adequate abstraction.

Timing behavior in RTOS's is coarse and becomes increasingly uncontrollable as the complexity of the system increases, e.g., by adding inter-process communication. Locks, priority inversion, interrupts and similar issues break the formalisms, forcing designers to rely on bench testing, which is nearly impotent at flushing out subtle timing bugs. Worse, these techniques produce brittle systems in which small changes can cause big failures. And as embedded systems become networked, the problems mount.

Synchronous digital hardware—the technology on which most computers are built—can deliver astonishingly precise, repeatable timing behavior, thanks in part to considerable efforts on the part of hardware designers and design tool builders. Software abstractions, however, lose several orders of magnitude in timing precision. Compare the nanosecond-scale precision with which hardware can raise an interrupt request to the imprecision with which a user-level software thread sees the effects (perhaps milliseconds).

Commercial RTOS's market predictable timing, but modern processors have rendered such numbers only vague bounds. Real-time software developers have long demanded predictable timing; processor architectures no longer deliver.

2 The Solution

It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function. We call them precision timed (PRET) machines. Our basic argument is that real-time systems, in which temporal behavior is as important as logical function, are an important and growing application; processor architecture needs to follow suit.

This is an enormous problem, but it is easy to start making progress. The problem is challenging because it spans nearly all abstraction layers in computing, including programming languages, virtual memory, memory hierarchy, pipelining techniques, power management, I/O, DRAM design, bus architectures, memory management, just-in-time (JIT) compilation, multitasking (threads and processes), task scheduling, software component technologies, and networking.

Our first step is to develop FPGA-targeted PRET cores suitable for high-reliability embedded applications. Substantial progress can be made in months; the revolution may take decades. Our ultimate goal is networked real-time software that delivers the reliability and timing precision of synchronous digital hardware with the simplicity of software.

Timing precision is easy to achieve if you are willing to forgo performance; the engineering challenge in PRET machines is to deliver both. While we cannot abandon structures such as caches and pipelines and 40 years of progress in programming languages, compilers, operating systems, and networking, many will have to be re-thought.

Fortunately, there is much work on which to build. ISAs can be extended with instructions that deliver precise timing with low overhead [7]. Scratchpad memories can be used in place of caches [1]. Deep pipelines with pipeline interleaving can deliver precise timing [10]. Memory management pause times can be bounded [2]. Programming languages can be extended with timed semantics [6]. Appropriately chosen concurrency models can be tamed with static analysis [3]. Software components can be made intrinsically concurrent and timed [11]. Networks can provide high-precision time synchronization [8]. Schedulability analysis can provide admission control, delivering run-time adaptability without timing imprecision [4].

Our vision of a mature PRET machine incorporates most of these techniques. At the ISA level, it provides cycle-accurate timers, a predictable memory hierarchy based on scratchpad memories, and an interleaved pipeline that provides predictable hardware-efficient concurrency. It will be programmed in a C-like language that includes user-specified timing constraints and concurrency, probably through a coordination language, perhaps with synchronous semantics. Both compile- and run-time checks will ensure the program meets timing constraints, similar to array bounds checking. A PRET operating system will resemble an RTOS, but its scheduling policies will provide guarantees and admission control. Such a processor will communicate through a network able to provide timing guarantees, probably leveraging time synchronization.

Many open challenges remain. How do we achieve high-precision I/O (classical interrupts destroy all temporal predictability)? How do we manage disk systems, DRAM behavior, and virtual memory? How do we scale to deep sub micron without losing the precision of synchronous digital logic (see http://www.tauworkshop.com)? How do we adapt operating systems to provide timing *guarantees*? How do we handle exceptions? How do we handle variable clock rates (essential power management)? How do we get precise timing in networking? How do we evolve the many fledgling research results into mainstream software engineering?

PRET machines are essential for embedded systems, but are also valuable for general-purpose systems. In concurrent software, non-repeatable behavior is a major obstacle to reliability [9]. PRET machines would improve reliability of concurrent software through repeatable concurrent behavior.

Patterson and Ditzel's [12] plea for RISC machines was simultaneously heeded and ignored. Architectural complexity continued to grow unabated, but at least architects began to analyze where it would have the most benefit. It forced architects to evaluate the benefits of their elaborations relative to the costs. A similar change is needed with respect to techniques that blithely ignore predictable timing.

References

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

[2] D. F. Bacon, P. Cheng, and V. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, Catania, Sicily, 2003.

[3] G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003. Overview of SCADE/Lustre.

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

[5] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, volume 2211 of *LNCS*, pages 469–485, North Lake Tahoe, California, 2001. Springer.

[6] 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.

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

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

[9] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[10] E. A. Lee and D. G. Messerschmitt. Pipeline interleaved programmable dsps: Architecture. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-35(9), 1987.

[11] E. A. Lee, S. Neuendorffer, and M. J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers*, 12(3):231–260, 2003.

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

[13] N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.