Precision Timed Infrastructure

Promoting Time to a First-Class Citizen in System Design

David Broman[*], **Stephen A. Edwards**[†], **Edward A. Lee**[‡]
[*]University of California, Berkeley and Linköping University, broman@eecs.berkeley.edu
[†]Columbia University, sedwards@cs.columbia.edu
[‡]University of California, Berkeley, eal@eecs.berkeley.edu

I. The Need for Precise Control of Time

In a cyber-physical system (CPS) [8], timing contributes to correctness, not just performance. Better average-case performance may improve the user experience, but consistently meeting deadlines may be key to safe behavior. Yet most programming languages, such as C or C++, provide no control over timing. The execution time of software is a complex, brittle function of both the software itself and the hardware on which it runs [15].

As a consequence, hard real-time systems are not portable. Costly testing, verification, and certification must consider the details of how software interacts with the hardware; any change in the hardware or software can have unpredictable effects on timing, forcing all this work to be repeated. For example, a small change in a cache replacement policy could lead to thrashing in an inner loop and much slower execution.

Software timing is brittle even on a single platform; caches, branch predictors, and complex pipeline interactions enable small code changes to strongly affect global timing. And it is not just the the program's source code; using a compiler optimizer, changing a linker, or changing the operating system's scheduling policy can cause big changes in timing.

Modeling languages for CPS have long recognized the need to precisely model time and treat it as a first-class entity. Modelica [11], Simulink [10], and Ptolemy [6] can precisely specify and simulate the timing behavior of both the physical and computational (cyber) parts.

Many of these modeling environments are even able to compile models into C or similar low-level platform-dependent code, but few execution platforms are able to guarantee the timing behavior of the generated code. This is regrettable; designers carefully specify and analyze the timing behavior of their systems, yet existing implementation schemes essentially discard this and force designers to validate the timing behavior of their implementations through testing.

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), #0931843 (ActionWebs), and #1035672 (CSR-CPS Ptides)), the U. S. Army Research Laboratory (ARL #W911NF-11-2-0038), the Air Force Research Lab (AFRL), the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program, and the following companies: Bosch, National Instruments, Thales, and Toyota. The first author was funded by the Swedish Research Council.

We believe both hardware and software platforms must change to provide timing that can be controlled as precisely as logical function.

II. Hardware Considerations

Modern computer architecture focuses on increasing overall performance, often at the expense of worst-case behavior or increased unpredictability. Deeper pipelines improve throughput at the expense of increasing the amount of time it takes to flush and restart a pipeline when a branch is mispredicted. Branch predictors reduce the probability of misprediction at the expense of additional, largely unpredictable, architectural state. Caches can greatly reduce average memory latency, but often increase the latency of a miss. Complex cache replacement strategies improve performance but, again, increase unpredictable architectural state.

Predictability is easy to achieve by itself—most microprocessors from the 1970s and 80s were completely predictable because they were very simple (and fairly low-performance); the real challenge is making a high-performance predictable processor.

In part due to our proposal for precision times (PRET) machines [5], we and others have been developing predictable hardware platforms [9], [13], [1]. In our own work [4], [9], we used a thread-interleaved pipeline (to avoid difficult-to-analyze pipeline hazards), a predictable DRAM controller, and scratchpad memories [2] instead of caches. We showed such an approach can greatly simplify worst-case execution time analysis.

But much work remains to be done. Main memory (DRAM) latency, which can be hundreds of cycles in today's technology, is a fundamental stumbling block. Any reasonably high-performance processor must store frequently accessed data in a smaller memory such as a cache or a scratchpad, yet caches greatly increase the amount of architectural state that must be tracked to predict the execution time of sequences of code.

Software control of a memory hierarchy seems necessary for reasonable performance and predicability, yet doing so using classical techniques such as DMA transfers seems likely to add significant performance overheads because it would add management code to the software's critical path. While some mixed alternatives have been proposed, such as Whitham and Audsley's scratchpad MMU [14], the problem is hardly considered solved.

III. Software Considerations

Merely providing hardware with predictable timing behavior is not enough—the software toolchain must support time as a first-class entity throughout. We call this solution—where time is a *correctness criterion*—a precision timed (PRET) infrastructure.

At the lowest level of abstraction, designers need to be able to specify timing behavior and expect it to be obeyed as precisely as arithemtic is now. We would never accept a software environment that thinks $1+1=3$; why should we accept one that allows $1\,\mathrm{s}+1\,\mathrm{s}=3\,\mathrm{s}$?

Bui et al. [3] describe a software construct called *meet the final deadline* (MTFD) that has this character. A code block is assigned a deadline and the program will refuse to run if it cannot meet the deadline. At lowest level of abstraction, MTFD together with get time (GT) and delay until (DU) are realized as processor instructions, as illustrated in the following extended ARM ISA assembly code example.

```
1  gt      r1, r2      Get time in ns (64 bits)
2   ...computation... Perform computation
3  ldr     r3, =10000  Increase the timer with 10us
4  adds    r2, r2, r3
5  adc     r1, r1, #0
6  mtfd    r1, r2      Takes at most 10us
7  du      r1, r2      Delay until, takes at least 10us
```

Timing instructions are shown in bold. Note how the absolute time (line 1) is incremented with the relative deadline of $10\mu s$ (lines 3-5). MTFD (line 6) specifies an upper bound of the execution time and delay until (line 7) makes sure that the computation takes at least $10\mu s$.

The software toolchain must check such timing instructions and provide guarantees. If MTFD instructions appear in the executable code as shown, the linker-loader may reject a program if it can't guarantee the timing. This approach is inspired by Necula's proof carrying code [12], where the verification process is divided into an offline certification stage followed by an online validation stage.

Static verification of MTFD constraints means computing a safe upper bound of WCET and comparing it to the constraint. Traditionally, the main challenge of WCET analysis is to compute a tight upper bound, which includes both loop bound detection, infeasible path detection, and low level machine timing analysis [15]. Recent work on WCET-aware compilation [7], on the other hand, utilizes compiler optimization phases to minimize WCET instead of the average case execution time. We propose a compiler that attempts to minimize the average execution time of blocks that are *not* constrained. That is, we view the compiler optimization problem as a traditional compiler problem with constraints on the MTFD blocks. Hence, the challenge is not to minimize WCET, but to make WCET bounds tight *and* close to MTFD.

Various modeling languages have different ways of expressing timing constraints. Creating a new compiler for each modeling language—with precision time capabilities—is impractical. We propose instead to define an intermediate language to which various modeling languages may be compiled. Such an intermediate language must have a well-defined semantics that encompasses both function and timing. At the highest level of abstraction, our intermediate language will take the form of a timed extension to C, called Precision Timed C (ptC). For example, here is a loop for a periodic control system in ptC:

```
int first = 1;
loopin(20ms){
  if(!first) actuate_data();
  first = 0;
  sense_data();
  control_computation();
}
```

Actuation takes place every 20 ms: an explicit constraint.

Although the exposed timing abstractions of ptC make it easier for various modeling languages to be compiled with precision time, the intermediate language would be only marginally useful if it also exposed all details of the PRET machines. In particular, scratchpad allocation schemes must be abstracted away, yet static or dynamic memory management decisions have a profound affect on WCET analysis and thus also for MTFD. It follows that a key challenge for the PRET infrastructure is to design the toolchain such that MTFD constraints are guaranteed to hold, average case performance of non MTFD blocks are optimized, and limited memory resources are utilized efficiently.

References

[1] S. Andalam, P. Roop, and A. Girault. Predictable multithreading of embedded applications using PRET-C. In *Proc. MEMOCODE*, pages 159–168, 2010.

[2] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proc. CODES*, pages 73–78, 2002.

[3] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke. Temporal isolation on multiprocessing architectures. In *Proc. Design Automation Conf.*, San Diego, CA, 2011.

[4] S. A. Edwards, S. Kim, E. A. Lee, I. Liu, H. D. Patel, and M. Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proc. ICCD*, pages 54–59, Oct. 2009.

[5] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Proc. Design Automation Conf.*, pages 264–265, June 2007.

[6] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, January 2003.

[7] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Real-Time Systems*, 46(2):251–300, 2010.

[8] E. A. Lee. Cyber physical systems: Design challenges. In *Intl. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369, 2008.

[9] I. Liu, J. Reineke, and E. A. Lee. A PRET architecture supporting concurrent programs with composable timing properties. In *Asilomar Conf. on Signals, Systems, and Computers*, November 2010.

[10] MathWorks. The Mathworks—Simulink—Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/.

[11] *Modelica—A Unified Object-Oriented Language for Physical Systems Modeling—Language Specification*, 2012. http://www.modelica.org.

[12] G. C. Necula. Proof-carrying code. In *Proc. Principles of Programming Languages*, pages 106–119, New York, USA, 1997.

[13] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54(1–2):265–286, 2008.

[14] J. Whitham and N. Audsley. Implementing time-predictable load and store operations. In *Proc. Embedded Software (Emsoft)*, pages 265–274, Grenoble, France, Oct. 2009.

[15] R. Wilhelm et al. The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embedded Computing Systems*, 7:36:1–36:53, May 2008.

DAVID BROMAN

University of California, Berkeley and Linköping University
broman@eecs.berkeley.edu, 510–460–0280

David Broman is currently a visiting scholar at UC Berkeley, USA, working in the Ptolemy group at the Electrical Engineering & Computer Science department. He is an assistant professor at Linköping University in Sweden, where he also received his Ph.D. in computer science in 2010. David's research interests include programming and modeling language theory, compiler technology, software engineering, and mathematical modeling and simulation of cyber-physical systems. He has worked five years within the software security industry, co-founded the EOOLT workshop series, and is member of the Modelica Association and the Modelica language design group.

STEPHEN A. EDWARDS

Columbia University,
sedwards@cs.columbia.edu, 212–939–7019

Stephen A. Edwards received the B.S. degree in Electrical Engineering from the California Institute of Technology in 1992, and the M.S. and Ph.D degrees, also in Electrical Engineering, from the University of California, Berkeley in 1994 and 1997 respectively. He is currently an associate professor in the Computer Science Department of Columbia University in New York, which he joined in 2001 after a three-year stint with Synopsys, Inc., in Mountain View, California. His research interests include embedded system design, domain-specific languages, compilers, and high-level synthesis.

EDWARD A. LEE

University of California, Berkeley,
eal@eecs.berkeley.edu, 510–643–3728

Edward A. Lee is the Robert S. Pepper Distinguished Professor and former chair of the Electrical Engineering and Computer Sciences (EECS) department at U.C. Berkeley. His research interests center on design, modeling, and simulation of embedded, real-time computational systems. He is a director of Chess, the Berkeley Center for Hybrid and Embedded Software Systems, and is the director of the Berkeley Ptolemy project. He is co-author of five books and numerous papers. He has led the development of several influential open-source software packages, notably Ptolemy and its various spinoffs. His bachelors degree (B.S.) is from Yale University (1979), his masters (S.M.) from MIT (1981), and his Ph.D. from U.C. Berkeley (1986). He is a Fellow of the IEEE, was an NSF Presidential Young Investigator, and won the 1997 Frederick Emmons Terman Award for Engineering Education.