Precision Timed Infrastructure: Design Challenges

(Invited Paper)

David Broman[*][†], Michael Zimmer[*], Yooseong Kim[‡], Hokeun Kim[*], Jian Cai[‡],
Aviral Shrivastava[‡], Stephen A. Edwards[§], and Edward A. Lee[*]
[*]University of California, Berkeley, {broman, mzimmer, hokeunkim, eal}@eecs.berkeley.edu
[†]Linköping University, david.broman@liu.se
[‡]Arizona State University, {yooseong.kim, jian.cai, aviral.shrivastava}@asu.edu
[§]Columbia University, sedwards@cs.columbia.edu

***Abstract*—In general-purpose software applications, computation time is just a quality factor: faster is better. In cyber-physical systems (CPS), however, computation time is a correctness factor: missed deadlines for hard real-time applications, such as avionics and automobiles, can result in devastating, life-threatening consequences. Although many modern modeling languages for CPS include the notion of time, implementation languages such as C lack any temporal semantics. Consequently, models and programs for CPS are neither portable nor guaranteed to execute correctly on the real system; timing is merely a side effect of the realization of a software system on a specific hardware platform. In this position paper, we present the research initiative for a precision timed (PRET) infrastructure, consisting of languages, compilers, and microarchitectures, where timing is a correctness factor. In particular, the timing semantics in models and programs must be preserved during compilation to ensure that the behavior of real systems complies with models. We also outline new research and design challenges present in such an infrastructure.**

I. Introduction

In a cyber-physical system (CPS) [23], timing contributes to correctness, not just performance. Better average-case performance may improve the user experience, but consistently meeting deadlines may be crucial to safe behavior. Yet most programming languages, such as C or C++, provide no direct control over timing. The execution time of software is a complex, brittle function of the software itself and the hardware and software environment in which it runs [35].

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.

On typical hardware platforms, caches, branch predictors, and complex pipeline interactions enable small code changes to strongly affect global timing. And the change does not have to occur at the source code level; changes in the compiler's level of optimization, the linker, or the operating system's scheduling policy can all cause big changes in timing.

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 (CPS: Large: ActionWebs), and #1035672 (CPS: Medium: Ptides)), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota. The first author was funded by the Swedish Research Council #623-2011-955.

Modeling languages and environments for CPS have long recognized the need to precisely model and control time. Modelica [30], Simulink [28], and Ptolemy II [12] can precisely model time in both 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 re-validate the timing behavior of their implementations through testing.

We believe both software and hardware platforms must fundamentally change for timing to be controlled as precisely as logical functionality. This idea is not new. In 2007, Edwards and Lee [10] proposed *precision timed (PRET) machines*—a new era of processors where temporal behavior is as important as logical functions. In this paper, however, we consider the whole software and hardware stack, including modeling and programming languages, compilers, and processors, where time is first-class citizen. The overall problem is to automatically compile or synthesize a model's cyber parts, such that the simulated model and the real system coincide. The key challenge of this *model fidelity* problem is to guarantee correct timing behavior [4]. We call a correct-by-construction solution to this problem—where time is a correctness criterion—a *precision timed (PRET) infrastructure*. Such an infrastructure should include three key components:

- *Precision timed languages*, programming or modeling languages where time is a first-class citizen—an integral part of the language's semantics. Specifically, we emphasize the need for a *precision timed intermediate language* that is independent of the source language. Such an intermediate language must hide low-level implementation details (such as the memory hierarchy) but still expose primitives for expressing timing (Section II).
- *Precision timed hardware*, physical components with predictable timing behavior. Typical processors sacrifice predictability to improve average-case performance; PRET processors and memory controllers attempt to

regain predictability while maintaining performance. The PRET machine language specified by the instruction set architecture (ISA) must also allow for greater control of timing behavior (Section III).

- *Precision timed compilers*, which preserve timing semantics as they translate from a PRET language to a PRET machine language. The challenge for PRET compilation is to guarantee the semantics of a high-level language and the timing behavior of execution on a specific hardware platform will coincide (Section IV).

II. Precision Timed Languages

To enable portability and correctness-by-construction, languages for modeling or implementing CPS need to include time as part of the programming model. In this section, we discuss what abstractions such languages should include. In particular, we motivate and discuss the need for timed intermediate languages.

A. Language Hierarchy

Various modeling languages have different ways of expressing computations and timing constraints [5]. For instance, Modelica [30], Simulink [28], Giotto [17], Ptolemy II [12], and Modelyze [6] all have different semantics for expressing time[1]. Creating a new compiler for each such modeling language—with precision time capabilities—is, however, impractical. We propose instead to define a common *intermediate language* to which various modeling languages may be compiled. Such an intermediate language must have a well-defined semantics that encompass both function and timing.

Figure 1 depicts how a timed intermediate language could act as a language-independent layer between modeling and programming languages and microarchitectures. The ovals at the top of the figure exemplify various modeling languages, and the arrows show compilation steps. Target languages for compilation are either a low-level implementation language (e.g., C extended with timing constructs[2]) or a low-level precision timed intermediate language (PRETIL).

The purpose of a PRETIL is to expose timing primitives to upper layers while abstracting away hardware details. In particular, memory hierarchies (cache or scratchpad memories), specific instruction set architectures (ISA), and hardware register allocation should be hidden from layers above the intermediate language. Instead, the intermediate language provides named variables, constructs for allocating memory, and other ISA-independent instructions. All these abstractions and constructs are common in standard intermediate representations (IR), such as LLVM's [22] IR. The unique difference

[1]Ptolemy II (using directors) and Modelyze (using embedded domin-specific languages) can express various models of computation, each with different timing semantics.

[2]Note how a C language with timing constructs is here viewed as an intermediate language and not as an implementation language. Today, this is already very common; many modeling environments use C as their target language for code generation. Existing timed C-based implementation languages, such as Real-time concurrent C [15] or Real-time Euclid [20], could potentially also be used in this infrastructure as long as they can compile to PRETIL.

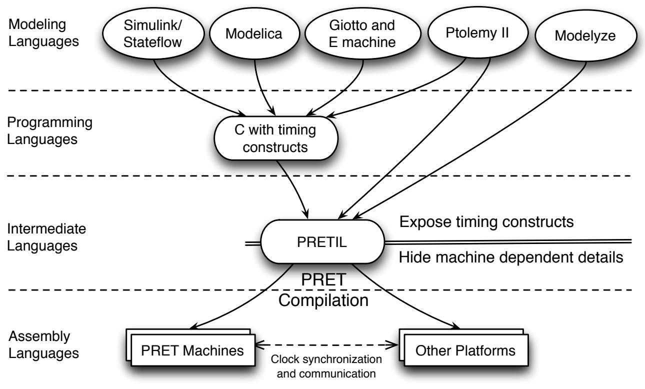

Fig. 1. Conceptual overview of translation steps between languages in a PRET infrastructure. At the top, various modeling languages are compiled into a precision timed intermediate language (PRETIL), potentially via a timed C programming language. The compilation step between a PRETIL (Section II) and PRET machines (Section III) is performed by the PRET compiler (Section IV). Programs and models may ideally be compiled and partitioned into several distributed platforms.

of a PRETIL is to include constructs for reasoning about *time*. By time we mean *real-time*, that is, execution time expressed at a certain precision, such as nanoseconds.

B. Mechanisms for Detecting Missed Deadlines

Embedded systems that interact with the physical environment through sensors and actuators need to meet timed *deadlines*. Relative deadlines are associated with *tasks*, functions, or program fragments that need to be executed at a certain time. A task may be classified according to the consequences of missing a deadline [8]: missing a *hard* task's deadline is catastrophic; the result of a *firm* task is useless after its deadline, but missing it is harmless; finally, the utility of the result of a *soft* task merely drops if its deadline is missed.

A precision timed language should be able to handle all three kinds of tasks. Figure 2 illustrates the three different scenarios. For soft tasks, a PRETIL language should provide *late miss detection* that can indicate after a task completes whether its deadline was missed and by how long. The program can use this information to make runtime decisions of how to handle missed deadlines. Example of soft tasks are tasks involved in user interactions or system logging. Firm tasks should be provided *immediate miss detection* that throws an exception exactly when a task's deadline expires. Firm tasks can be found in multimedia and networking applications; missing a frame or packet are less important than introducing latencies. Another use for immediate miss detection is *anytime algorithms*; that is, algorithms that produce better results the longer they are executed and return a valid solution when interrupted. Finally, hard tasks need *early miss detection* that indicate long before the task runs whether it will miss its deadline. Hard tasks are typically found in safety critical systems such as sensor data acquisition systems, image processing, and control systems. Late and immediate misses may be detected at runtime, but early miss detection requires static analysis; an upper bound

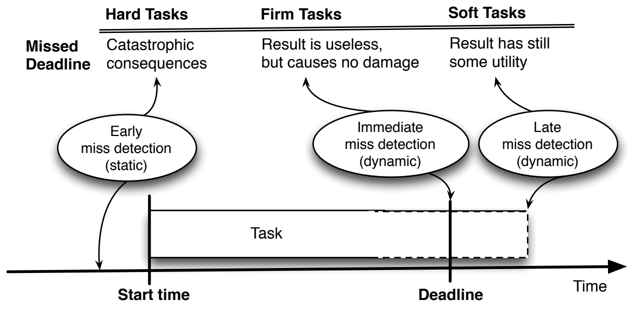
Hard Tasks
Firm Tasks
Soft Tasks
Missed Deadline
Catastrophic consequences
Result is useless, but causes no damage
Result has still some utility
Early miss detection (static)
Immediate miss detection (dynamic)
Late miss detection (dynamic)
Task
Start time
Deadline
Time

Fig. 2. Relationship between types of tasks and deadline detection.

of the worst-case execution time (WCET) [35] must be less or equal to the relative deadline. Languages that are part of a precision timed infrastructure should—with a certain precision—include semantics for handling all these kinds of detection mechanisms.

C. Towards a PRET Intermediate Language

Our current work-in-progress is focusing on extending the *low level virtual machine (LLVM)* [22] assembly language with timing semantics. We call this precision timed extension *ptLLVM*.

Timing constraints for hard real-time tasks, which require early miss detection, can be expressed using a software construct called *meet the final deadline* (MTFD), originally proposed as a hardware instruction [7]. A code block is assigned a deadline and the program will refuse to compile if it cannot meet the deadline. As input the compiler needs, besides the timed program, a specification of the target architecture. If the target microarchitecture is a PRET machine (see bottom part of Figure 1), verification of deadlines is significantly simplified (see Section IV) compared to a standard general purpose processor. A PRET intermediate language is not, however, in principle required to be compiled to a PRET machines; compilation could also be done for a standard embedded platform with timer support—although the precision of time may not be the same.

The following simple ptLLVM code shows how a program can be given an upper and a lower time bound.

```
1  %t1 = gt i64              ; Get current time in ns
2  mt i64 10000              ; Takes at most 10us
3   ; ...computation...
4  %t2 = add i64 %t1, 5000
5  du i64 %t2                ; Takes at least 5us
6  fd
```

The example illustrates four new timing instructions[3], shown in bold. On line 1, instruction **gt** (get time) returns a 64-bit integer value (i64) representing number of nanoseconds elapsed since system boot time. The returned value is stored in the local variable `%t1`. The MTFD construct is divided into two separate instructions; **mt** (line 2) specifying the beginning of the timing constrained block and **fd** (line 6) specifying the end. The MTFD block states that all code between **mt** and **fd** must execute within $10\mu s$. As a consequence, this MTFD block gives a static *upper bound* of the execution time. The software toolchain must check such timing instructions and provide guarantees, which is further discussed in Section IV. Variable `%t2` is assigned an incremented value, which is used by the **du** (delay until) instruction (line 6) to express an lower bound of 5 μs. Consequently, the execution time of the computation task (line 3) is bound to be between 5 μs and 10 μs: an explicit time constraint.

Although not shown in the above example, one more instruction is needed to cover all different variants of deadline miss detection. To perform immediate miss detection, we introduce an instruction called **ee** (exception on expire). The purpose of this instruction to specify a timed exception that is raised exactly when a specified deadline is missed.

D. PRET Language Design Challenges

A number of questions arise while designing a language hierarchy with a ubiquitous notion of time:

- What is the minimal set of timing constructs of an intermediate language for expressing real-time semantics of arbitrary modeling languages? In our current work, this set consists of the "get time," "delay until," "exception on expire," and "meet the final deadline" instructions. We believe this is enough in a single threaded setting; comprehensive studies showing how to compile several modeling languages could help to confirm this.
- How should the precision of time be expressed? Most modeling languages simply assume perfect clocks and timing. However, it matters whether the implementation platform can deliver millisecond- or nanosecond-level precision. The behavior of the realized system may substantially differ depending on the precision.
- How should concurrency be expressed in a timed intermediate languages? In several formalisms and languages, such as Kahn process networks [19] or synchronous languages [3], concurrency is an inherent property. The target platform may be parallel, having multiple cores, hardware threads, or both. Yet most implementation languages (e.g., C) and intermediate assembly representations (e.g., LLVM) do not provide explicit concurrency constructs. The challenge is to provide both constructs for expressing concurrency and to provide predictable communication mechanisms.
- What is the role of a real-time operating system (RTOS) for a precision timed infrastructure? Should scheduling of tasks be part of a RTOS (designed for PRET languages) or should the system be bare metal and scheduling be part of the compilation process?
- How can timed intermediate languages be compiled in a distributed setting? If the modeling language is based on a distributed model of computation, such as PTIDES [36], we would like the infrastructure to partition the implementation into distributed pieces.

[3]The timing instructions are here shown, for brevity, as native LLVM instructions, but are in our current work-in-progress defined as intrinsic functions.

III. Precision Timed Hardware

Modern computer architectures focus on increasing overall application performance, often at the expense of the performance and predictability of individual operations. Deep pipelines with branch predictors and caches improve average-case performance when high-penalty "misses" are seldom. Unfortunately, the same principle that improves average-case performance—using execution history to predict future operation—is a main source of unpredictability.

Predictability is easy to achieve by itself—most microprocessors from the 1970s and 80s were completely predictable because they were simple and fairly low-performance as a result; the real challenge is making a high-performance predictable processor. In part due to our proposal for precision times (PRET) machines [10], we and others have been developing predictable hardware platforms [1], [27], [33].

A. Pipelines

A technique to maintain performance without branch prediction is to use a thread-interleaved pipeline [24], as done in our previous work [9], [26], [27]. Each pipeline stage contains an instruction from a different hardware thread. Branches are resolved before a thread's next instruction starts, so no prediction is required and the pipeline stays full. Each hardware thread has a lower throughput, but the processor is more predictable and has higher overall throughput.

Processor cycles are wasted, however, if limited concurrency within the system results in not all hardware threads being utilized. We are currently developing a more flexible PRET processor to improve performance and maintain timing predictability for varying concurrency by using a predictable, software-controlled, thread interleaving scheduler. Cycles that would otherwise be wasted can be used by a different hardware thread, but dependencies between instructions are reintroduced and require forwarding or stalling to prevent hazards.

B. Memory

Caches can greatly reduce average memory latency, but predicting whether a memory access will hit or miss requires knowledge of the current cache state. In our previous work, we use scratchpad memories (SPM) [2] instead of caches. For programs that cannot fit in the scratchpad memory, a predictable DRAM controller [32] may be used for accessing a larger main memory. In a SPM, explicit instructions are required to move data from slower, main memory into the faster, local scratchpad memory; this is typically done as a DMA transfer from DRAM to SPM.

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 predictability, yet doing so using classical techniques such as DMA transfers is 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 memory management unit [34], the problem is hardly considered solved.

C. Instruction Set Architecture

Time cannot be explicitly controlled in modern *instruction set architectures* (ISA), only indirectly controlled with software and available hardware, such as tick counters. Consequently, the timing behavior of a binary program is platform-dependent: it depends on available hardware and its configuration.

Previous work [25] extends an ISA to provide direct control over timing. A real-time clock accessible to software with dedicated instructions, as opposed to software interacting with a tick counter, enables timing behavior to both be specified in a binary program and have higher precision with less overhead. In future work, the clock could also be synchronized to other platforms or a more accurate clock source using a clock synchronization method, such as IEEE 1588 [11].

D. PRET Hardware Challenges

Below are a few of the questions we foresee arising in the development of PRET hardware.

- How best should software manage the memory hierarchy? Current cache-based architectures almost entirely hide such management, improving programmability at the expense of predictability. In effect, a very complicated cache management "program" is always running in parallel with user code; understanding the temporal behavior of the pair require detailed knowledge of both and how they interact. PRET hardware is likely to both change and expose the memory management system, but what, exactly, should this look like? One extreme is for the programmer to control all DMA transfers, such as is done in the CELL processor [18], but such an architecture is difficult to program. A satisfactory solution to this problem may be key to a practical PRET architecture.
- How should the pipelines be structured? Purely thread-interleaved pipelines guarantees non-interference at the expense of latency. The behavior of aggressive best-effort pipelines is complex and depend strongly on the interaction of nearby instructions. Is there a happy medium in which non-interacting instructions from the same thread could proceed more quickly through a multi-stage pipeline without sacrificing predictability? Should each thread be provided different performance and predictability guarantees, as in a mixed-criticality system?
- How should timing specification and control be added to the ISA? Direct access to a real-time clock and the ability to react to it "instantly" seems like a bare minimum, but should the ISA provide more control over, say, instruction scheduling? Should it prescribe instruction-level timing that the microarchitecture must obey?

IV. Precision Timed Compilation

In addition to the traditional roles of expressing the application in terms of machine instructions and improving overall performance, the goal of a precision timed compiler is to ensure that the hard timing constraints imposed by MTFD constructs are met. Traditionally, compilation and verification of timing constraints have been completely separate tasks. The design process in such an approach is a repeated loop of compilation and testing until timing constraints are met or considered infeasible. The compilation process has no timing models for execution, even though the compiler has a significant impact on an application's execution time. This complex design loop can be performed in a much more intelligent and integrated fashion by adding the timing analysis capabilities inside the compiler. We call such a compiler a *PRET compiler*.

A. Compiling for Parameterized Microarchitectures

A major change that PRET compilation introduces is that compilation is targeting a specific microarchitecture for an ISA rather than the purely functional ISA itself. This is because the execution time of an application depends strongly on the processor microarchitecture and far less on the ISA. Unfortunately, dependency on the microarchitecture reduces the portability of object code. To resolve this portability challenge, PRET compilers need to operate on parameterized processor architectures, represented by, for example, an *architecture description language (ADL)*, such as EXPRESSION [29]. In such a case, all the timing parameters must be described in the ADL, which are used to estimate the timing of applications.

B. Worst-Case Execution Time Analysis

Static verification of MTFD constraints means computing a safe upper bound on WCET and comparing it to constraints. Traditionally, the main challenge of WCET analysis [35] is to compute a tight upper bound, which includes both loop bound detection [21], infeasible path detection [16], and low level machine timing analysis [14]. More recent work on WCET-aware compilation [13] 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 constraints.

An alternative is to delay the MTFD constraint verification to load time. Such an approach would resemble Necula's proof carrying code [31], where the verification process is divided into an offline certification stage followed by an online validation stage.

Another major challenge of WCET analysis for non-PRET architectures is interference due to resource sharing. Consequently, temporal isolation or bounding interference are important requirements for WCET analysis. Temporal isolation is also important for composability—a necessary property for scalability.

C. Scratchpad Memory Allocation

Although the exposed timing abstractions of the intermediate language 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 effect on WCET analysis and thus also for MTFD constraints. 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.

By using scratchpad memories, the compiler has substantial control over timing, but also the extremely challenging task of meeting all, potentially competing, timing constraints. Therefore, key challenges are to develop techniques for managing code and data on SPMs, computing safe bounds of WCET for different management schemes, and selecting code and data management schemes that meet and balance timing constraints.

D. PRET Compiler Challenges

The questions that arise in the development of a PRET compiler chain depend, in large part, on decisions made during the design of the source languages and target hardware.

- How microarchitecture-aware must a PRET compiler be? Does it need full information about the details of the pipeline, or is there a way to instead have the compiler dictate, say, scheduling information through the ISA that the microarchitecture would then obey? Modern processors have rendered existing ISAs fictional models of the underlying microarchitectures. Should a PRET compiler target a different, timing-aware, fiction, or should it be presented with an ISA that hides nothing?
- What balance should we strike between ahead-of-time and just-in-time compilation? Should a PRET executable be characterized by a simple clock-rate constraint (i.e., that ensures timing behavior provided the underlying processor has a sufficiently fast clock)? Should the platform validation operation be much more detailed, e.g., mimicking Necula's proof-carrying code [31]? Or should it take a just-in-time compilation approach in which the compiler actually adapts an executable to a particular platform? As usual, the question is a choice of which abstractions the ISA should present.
- How platform-aware must the compiler be? Few existing compilers take into account, say, a platform's cache configuration, but this may be desired in a PRET compiler. By contrast, existing compilers are always mindful (at some point in their operation) of the number and character of ISA-visible registers. A PRET just-in-time compiler may want to start with a very abstract model of the computation to be performed and tailor it to the details of the platform on which it will be executed, much like

how (stack-based) Java bytecodes are typically compiled onto register machines.

- How can optimization phases of the PRET compiler be aware of MTFD constraints? Must worst-case execution time analysis and compiler optimization be two different phases (potentially connected in a feedback loop), or can optimization and WCET analysis be combined into one integrated phase?

V. Conclusions

In this paper, we present a research initiative in which the notion of time is an integral part of the programming model. Our previous work focused on the hardware—PRET machines—whereas this work-in-progress considers the whole software stack, including precision-timed languages and compilers. Our proposed solution necessarily crosscuts multiple disciplines, especially the border between software and hardware; we contend that such multi-discipline system design is vital to achieve the final goal of an infrastructure with ubiquitous notion of time.

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] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[4] D. Broman. High-confidence cyber-physical co-design. In *Proceedings of the Work-in-Progress (WiP) session of the 33rd IEEE Real-Time Systems Symposium (RTSS 2012)*, page 12, 2012.

[5] D. Broman, E. A. Lee, S. Tripakis, and M. Törngren. Viewpoints, formalisms, languages, and tools for cyber-physical systems. In *Proceedings of the 6th International Workshop on Multi-Paradigm Modeling (to appear)*, 2012.

[6] D. Broman and J. G. Siek. Modelyze: a gradually typed host language for embedding equation-based modeling languages. Technical Report UCB/EECS-2012-173, EECS Department, University of California, Berkeley, June 2012.

[7] 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.

[8] G. C. Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. Springer, third edition, 2011.

[9] 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.

[10] 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.

[11] J. C. Eidson. *Measurement, Control, and Communication Using IEEE 1588*. Springer, 2006.

[12] 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.

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

[14] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2):131–181, 1999.

[15] N. Gehani and K. Ramamritham. Real-time concurrent C: A language for programming dynamic real-time systems. *Real-Time Systems*, 3(4):377–405, 1991.

[16] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS'06)*, pages 57–66. IEEE, 2006.

[17] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[18] IBM. *Cell Broadband Engine Architecture*, Oct. 2007. Version 1.02.

[19] G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 1974.

[20] E. Kligerman and A. D. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *Software Engineering, IEEE Transactions on*, 12(9):941–949, 1986.

[21] J. Knoop, L. Kovcs, and J. Zwirchmayr. Symbolic Loop Bound Computation for WCET Analysis. In E. Clarke, I. Virbitskaite, and A. Voronkov, editors, *Perspectives of Systems Informatics*, volume 7162 of *LNCS*, pages 227–242. Springer, 2012.

[22] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE, 2004.

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

[24] E. E. Lee and D. Messerschmitt. Pipeline interleaved programmable DSP's: Architecture. *Acoustics, Speech, and Signal Processing [see also IEEE Transactions on Signal Processing], IEEE Transactions on*, 35(9):1320–1333, 1987.

[25] I. Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012.

[26] I. Liu, J. Reineke, D. Broman, M. Zimmer, and E. A. Lee. A pret microarchitecture implementation with repeatable timing and competitive performance. In *To appear in Proceedings of International Conference on Computer Design (ICCD)*, October 2012.

[27] 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.

[28] MathWorks. The Mathworks - Simulink - Simulation and Model-Based Design. http://www.mathworks.com/products/simulink/ [Last accessed: May 8, 2013].

[29] P. Mishra, A. Shrivastava, and N. Dutt. Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):626–658, 2006.

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

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

[32] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee. PRET DRAM controller: Bank privatization for predictability and temporal isolation. In *CODES+ISSS*, pages 99–108. ACM, October 2011.

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

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

[35] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7:36:1–36:53, May 2008.

[36] Y. Zhao, J. Liu, and E. Lee. A programming model for time-synchronized distributed real-time systems. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS'07. 13th IEEE*, pages 259–268. IEEE, 2007.