# Computing Needs Time

*Edward A. Lee*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-30
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-30.html

February 18, 2009

Acknowledgement

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, HSBC, Lockheed-Martin, National Instruments, and Toyota.

# Computing Needs Time[*]

Edward A. Lee
UC Berkeley
`eal@eecs.berkeley.edu`

February 18, 2009

**Abstract**

This paper considers the orchestration of computing with physical processes. It argues that to realize its full potential, the core abstractions of computing need to be rethought to incorporate essential properties of the physical systems, most particularly the passage of time. It makes a case that the solution cannot be simply overlaid on existing abstractions, and outlines a number of promising approaches being pursued. The emphasis needs to be on repeatable behavior rather than on performance optimization.

## 1 Introduction

Most microprocessors today are embedded in systems that are not first-and-foremost computers. They are cars, medical devices, instruments, communication systems, industrial robots, toys, games, etc. Key to these microprocessors is their interaction with physical processes through sensors and actuators. Such microprocessors, however, increasingly resemble general-purpose computers. They are becoming networked and intelligent, often at the cost of dependendability. An acquaintance recently installed a wind turbine on his farm. “The wind turbine is up but not spinning,” he said.

[*]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, HSBC, Lockheed-Martin, National Instruments, and Toyota.

"It seems to have a computer; need I say more."[1] We have all heard such complaints. Whether the computer is culprit or not, they reflect a lack of confidence in the technology. Is that lack of confidence justified?

Even general-purpose computers are increasingly being asked to perform such interactions with physical processes. They integrate media such as video and audio, and through the migration to handheld platforms and pervasive computing systems, sense physical dynamics and control physical devices. Sadly, they don't do it very well. It is common today to establish a telephone connection that is of such poor quality that voices are incomprehensible. The first digital telephony systems deployed in the 1960s were better. The video quality that we routinely accept on the internet can be dramatically lower than the television broadcasts of the 1950s. We have entered a "Low-Fi" era.[2] The technological basis that we have chosen for general-purpose computing and networking does not match these applications well. Changes in this basis could improve these applications and enable many others.

The foundations of computing, rooted in Turing, Church, and von Neumann, are about the transformation of data, not about physical dynamics. This paper argues that we need to rethink the core abstractions if we really want to integrate computing with physical processes. In particular, I focus on a key aspect of physical processes that is almost entirely absent in computing, the passage of time. This is not just about "real-time systems," which accept the foundations and retrofit them with temporal properties. Although that technology has much to contribute, I will argue that it cannot solve the problem alone because it is built on flawed foundations.

Many readers will no doubt be already objecting. Computers have become so fast that surely the passage time in most physical processes is so slow that it can be handled without special effort. But then why is the latency of audio signals in modern PCs a large fraction of a second? Audio processes are quite slow by physical standards, and a large fraction of a second is an enormous amount of time. To achieve good audio performance in a computer (e.g. in a set-top box, which is required to have good audio performance), engineers are forced to discard many of the innovations of the last 30 years of computing. They often work without an operating system, without virtual memory, without high-level programming languages, without memory management, and without reusable component libraries, which

[1]Coonmessett Farm email newsletter, R. Smolowitz, June 26, 2008.

[2]P. Mosterman, in a talk at the Dagstuhl Seminar, Model-based Engineering of Embedded Real-time Systems, Nov. 5-9, 2007.

do not expose temporal properties on their interfaces. Those innovations are built on a key premise: that time is irrelevant to correctness; it is at most a measure of quality. Faster is better, if you are willing to pay the price. By contrast, what these systems need is not faster computing, but physical actions taken at the right time. It needs to be a semantic property, not a quality factor.

But surely the "right time" is expecting too much, the reader may object. The physical world is neither precise nor reliable, so why should we demand this of computing systems? Instead, we must make the systems robust and adaptive, building reliable systems out of unreliable components. While I agree that systems need to be designed to be robust, we should not blithely discard the reliability we have. Electronics technology is astonishingly precise and reliable, more than any other human invention. We routinely deliver circuits that will perform a logical function essentially perfectly, on time, billions of times per second, for years. Shouldn't we exploit this remarkable achievement?

I believe we have been lulled into a false sense of confidence by the considerable successes of embedded software, for example in automotive, aviation, and robotics applications. But the potential is vastly greater; we have reached a tipping point, where computing and networking may be integrated into the vast majority of artifacts that humans make. However, as we move to more networked, more complex, and more intelligent applications, the problems are going to get worse. Embedded systems will no longer be black boxes, designed once and immutable in the field. Instead, they will be pieces of a larger system, a dance of electronics, networking, and physical processes. An emerging buzzword for such systems is *cyber-physical systems* (CPS). The charter for the recent "CPS Summit" says[3]

> "The integration of physical systems and processes with networked computing has led to the emergence of a new generation of engineered systems: Cyber-Physical Systems (CPS). Such systems use computations and communication deeply embedded in and interacting with physical processes to add new capabilities to physical systems. These cyber-physical systems range from miniscule (pace makers) to large-scale (the national power-grid). Because computer-augmented devices are everywhere, they are a huge source of economic leverage."

[3]R. Rajkumar, B. Krogh, et al., *CPS Summit: Holistic Approaches to Cyber-Physical Integration*, April 24-25, 2008, St. Louis
http://ike.ece.cmu.edu/twiki/bin/view/CpsSummit/WebHome

> "... it is a profound revolution that turns entire industrial sectors into producers of cyber-physical systems. This is not about adding computing and communication equipment to conventional products where both sides maintain separate identities. This is about merging computing and networking with physical systems to create new revolutionary science, technical capabilities and products."

The challenge of integrating computing and physical processes has been recognized for some time [26], motivating the emergence of hybrid systems theories. Progress in that area, however, remains limited to relatively simple systems combining ordinary differential equations with automata. New breakthroughs are needed for CPS.

Applications of CPS arguably have the potential to rival the 20-th century IT revolution. They include high confidence medical devices and systems, assisted living, traffic control and safety, advanced automotive systems, process control, energy conservation, environmental control, avionics, instrumentation, critical infrastructure control (electric power, water resources, and communications systems for example), distributed robotics (telepresence, telemedicine), defense systems, manufacturing, and smart structures. It is easy to envision new capabilities that are technically well within striking distance, but that would be extremely difficult to deploy using today's methods. Consider, for example, a city without traffic lights, where each car provides the driver with adaptive information on speed limits and clearance to pass through intersections. We have in hand all the technical pieces for such a system, but achieving the requisite level of confidence in the technology seems decades off.

Other applications seem inevitable, but will be deployed without the benefit of many (or most) developments in computing. Consider distributed real-time games that integrate sensors and actuators to change the (relatively passive) nature of on-line social interactions. The engineering style of these systems will more closely resemble the set-top box than the PC.

I contend that today's computing and networking technologies *unnecessarily* impede progress towards these applications. In an article about "physical computing systems," Stankovic et al. [34] state "existing technology for RTES [real-time embedded systems] design does not effectively support the development of reliable and robust embedded systems." In this paper, I focus on the lack of temporal semantics. Today's "best effort" operating system and networking technologies cannot produce the levels of precision and reliability that most of these applications demand.

## 2 Glib Responses

Calling for a fundamental change in the core abstractions of computing is asking a lot. The reader may argue that the problems can be dealt with without such a revolution. To illustrate that this is not so, I examine some popular but misleading aphorisms, some of which suggest that incremental changes will suffice.

**"Computing takes time"**

This phrase is used to suggest that if only software designers would accept this fact of life, then the problems could be dealt with. However, the phrase is not using the commonly accepted meaning of the word "computing." In fact, computing is an abstraction of a physical process that takes time. But every abstraction omits some details (or it wouldn't be an abstraction), and one of the details that computing omits is time. This choice (to omit time) has been enormously beneficial, enabling the development of a very sophisticated technology. My main point in this paper is that there is a price. This choice has resulted in a mismatch with many of the applications to which we apply computing. Asking software designers to accept this fact of life, therefore, is asking them to forgo a key aspect of their most effective abstractions, without offering a replacement.

The term "computing" refers to the abstraction, not to the physical process. Were this not true, then a program in a programming language would not define a computation. One could only define a computation by describing the physical process. A computation is the same regardless of how it is executed. This is, in fact, the essence of the abstraction. When considering CPS, it is arguable that we (as a community) have chosen a rather inconvenient abstraction.

Moreover, the fact that the physical process takes time is only one of the reasons that the abstraction is inconvenient. It would still be inconvenient if the physical process were infinitely fast. In order for computations to interact meaningfully with other physical processes, they must include time in the domain of discourse.

**"Time is a resource"**

Computation, as expressed in modern programming languages, obscures many resource management problems. Memory is provided without bound by stacks and heaps. Power and energy consumption are not the concern of a programmer (mostly). Even when these resource management problems are important, there is no way to talk about them within the semantics of a programming language.

Time, however, is not quite like these other resources. First, barring metaphysical discourse, it is genuinely unbounded. To consider it a bounded resource, we would have to say that the available time per unit time is bounded, a tautology. Second, time gets expended whether we use it or not. It cannot be conserved and saved for later. This is true up to a point with, say, battery power, which is unquestionably a resource. Batteries leak, so their power cannot be indefinitely conserved, but designers rarely optimize a system to use as much battery power before it leaks away as they can. Yet that is what they do with time.

If time is a resource, it is a rather unique resource [20]. To lump together the problem of managing time with the problems of managing other more conventional resources will inevitably lead to the wrong solutions. Conventional resource management problems are optimization problems, not correctness problems. Using fewer resources is always better than using more. Hence, there is no need to make energy consumption a semantic property of computing. This is not true of time.

**"Time is a non-functional property"**

What is the "function" of a program? In computation, the function is a mapping from sequences of input bits to sequences of output bits (or an equivalent finite alphabet). The Turing-Church thesis defines "computable functions" to be those that can be expressed by a terminating sequence of such bits-to-bits functions, or mathematically by a finite composition of functions whose domain and codomain are the set of sequences of bits.

In CPS, the function of a computation is defined by its effect on the physical world. This is no less a function than a mapping from bits to bits. It is a function in the intuitive sense of "what is the function of the system," and can also be expressed as a function in the mathematical sense of a mapping from a domain to a codomain [18]. But as a function, the domain and codomain are not sequences of bits. Why are we insisting on the wrong definition of "function"?

Designers of operating systems, web servers, and communication protocols take a *reactive* view of programs, where a program is a sequence of input/output events rather than a mapping from bits to bits. This view needs to be elevated to the application programmer level and augmented with explicit temporal dynamics.

**"Real time is a quality of service problem"**

Everybody wants quality. Higher quality is always better than lower quality (at least, under constant resource usage, creating a paradox with "time is a resource"). Indeed, in general-purpose computing, a key quality measure is execution time (or equivalently, "performance"). But time in embedded systems plays a different role. Less time is not better than more time, as it is with performance. That would imply that it is better for an engine controller to fire the spark plugs earlier than later. Finishing early is not always a good thing, and in fact can lead to paradoxical behaviors where finishing early causes deadlines to be missed [8]. In an analysis that remains as valid today as 19 years ago, Stankovic [33] laments the resulting misconceptions that real-time computing "is equivalent to fast computing" or "is performance engineering." CPS requires *repeatable behavior* far more than optimized performance.

*Precision* and *variability* in timing are quality of service problems, but time itself is much more than that. If time is not present in the semantics of programs, then no amount of "quality of service" will adequately address timing properties of CPS.

## 3 Correctness

To solidify our discussion, we need to define some terms. Our definitions are based on the formal model known as the tagged signal model [19].

A *design* is a description of a system. For example, a C program is a design. So is a C program together with a choice of microprocessor, a choice of peripherals, and a choice of operating system. The latter design is more detailed (less abstract) than the former.

More precisely, a design is a set of behaviors. A *behavior* is a valuation of observable variables, including all externally supplied inputs. These variables may be themselves functions. For example, in a very detailed design, each behavior may be a trace of electrical signals at the inputs and outputs of the system. The *semantics* of a design is a set of behaviors.

In practice, a design is given in a *design language*, which may be formal, informal, or some mixture of the two. A design in a design language expresses the intent of the designer by defining the set of acceptable behaviors. Clearly, if the design language has precise (mathematical) semantics, then the set of behaviors is unambiguous. There could, of course, be errors in the expression, in which case the semantics will include behaviors that are not intended by the designer.

For example, a function given in a pure functional programming language is a design. We can define a behavior to be a pair of inputs and outputs (arguments and results). The semantics of the program is the set of all possible behaviors. This set defines the function specified by the program. Alternatively, we could define a behavior to include timing information (when the input is provided and when the output is produced). In this case, the semantics of the program includes all possible latencies (outputs can be produced arbitrarily later than the corresponding inputs), since nothing about the design language constrains timing.

A *correct execution* is any execution consistent with the semantics of the design. That is, given inputs, a correct execution finds a behavior consistent with those inputs in the semantics. If the design language has loose or imprecise semantics, then "correct" executions may be unexpected. Conversely, if the design expresses every last detail of the implementation, down to printed circuit boards and wires, then a correct execution may be, by definition, any execution performed by said implementation. For the functional program above, an execution is correct regardless of how long it takes to produce the output.

A *repeatable behavior* is a behavior exhibited by every correct execution given the same inputs in that behavior. For example, any behavior of the pure functional program is repeatable if we define behaviors without timing, but no behavior is repeatable if we define behaviors to include timing. How we define behaviors is important. The functional program can be made repeatable with timing by giving more detail in the design, for example by specifying a particular computer, compiler, and initial condition on caches, memory, etc. The design has to get far less abstract to make these behaviors repeatable.

A *predictable behavior* is a behavior that can be determined in finite time by analysis of the design. That is, given only the information expressed in the design language, it needs to be possible to infer the behavior given the inputs. For a particular functional program, behaviors may be predictable, but given an expressive enough functional language, it will always be possible to give programs where behaviors are not predictable. If the language is Turing complete, then behaviors may be undecidable. In practice, even "finite time" is not really sufficient. To be usefully predictable, behaviors need to be inferred in reasonable time.

Designs are generally abstractions of systems, omitting certain details. For example, even the most detailed design may not specify how behaviors change if the system is incinerated or crushed. An implementation of this design, however, does have specific reactions to these events (albeit probably not predictable reactions). *Reliability* is the extent to which an implementation of a design delivers correct behaviors over time and over varying operating conditions. A system that tolerates more operating conditions or remains correct for a longer period of time is said to be more reliable.

The operating conditions include conditions in the environment (temperature, input values, timing of inputs, humidity, etc.), but also may include conditions in the system itself, such as fault conditions (failures in communications, loss of power, etc.). A *brittle* system is one where small changes in the operating conditions or in the design yield incorrect behaviors. Conversely, a *robust* system remains correct with small changes in operating conditions or in the design. Making these concepts mathematically precise is extremely difficult for most design languages, so engineers are often stuck with intuitive and approximate assessments of these properties.

## 4 Requirements

Embedded systems have always been held to a higher reliability standard than general-purpose computing. Consumers do not expect their TV to crash and reboot. They have come to count on highly reliable cars, where in fact the use of computer controller has dramatically improved both the reliability and efficiency. In the transition to CPS, this expectation of reliability will only increase. In fact, without improved reliability, CPS will not be deployed into such applications as traffic control, automotive safety, and health care.

The physical world, however, is not entirely predictable. Cyber-physical systems will not be operating in a controlled environment, and must be robust to unexpected conditions and adaptable to subsystem failures. An engineer faces an intrinsic tension; designing reliable components makes it easier to assemble these components into reliable systems. But no component is perfectly reliable, and the physical environment will manage to foil reliability by presenting unexpected conditions. Given components that are reliable, how much can a designer depend on that reliability when designing the system? How does she avoid brittle designs?

The problem of designing reliable systems is not new in engineering. Two key engineering tools that we use are analysis and testing. Engineers analyze designs to predict behaviors under various operating conditions. For this to work, the designs must be predictable. They must yield to such analysis. Engineers also *test* systems under various operating conditions. Without repeatability, testing is a questionable practice.

Digital circuit designers have the luxury of working with a technology that delivers predictable and repeatable logical function *and timing.* This is true despite the highly random underlying physics. Circuit designers have learned to harness intrinsically stochastic physical processes to deliver a degree of repeatability and predictability that is unprecedented in the history of human innovation. In my opinion, we should be extremely reluctant to give this up.

The principle that we need to follow is simple. Components at any level of abstraction should be made as predictable and repeatable as is technologically feasible. The next level of abstraction above these components must compensate for any remaining variability with robust design.

Successful designs today follow this principle. It is (still) technically feasible to make predictable gates with repeatable behaviors that include both logical function and timing. So we design systems that count on this. It is harder to make wireless links predictable and repeatable. So we compensate one level up, using robust coding schemes and adaptive protocols.

The obvious question, therefore, is whether it is technically feasible to make software systems that yield predictable and repeatable behaviors for CPS. At the foundations of computer architecture and programming languages, software is essentially perfectly predictable and repeatable, if we limit the term "software" to refer to what is expressed in simple programming languages. Given an imperative language with no concurrency, well-defined semantics, and a correct compiler, designers can count on any computer with adequate memory to perform exactly what is specified in the program with nearly 100% confidence.

The problem arises when we scale up from simple programs to software systems, and particularly to CPS. The fact is that even the simplest C program is not predictable and repeatable in the context of CPS because *the design does not express aspects of the behavior that are essential to the system*. It may execute perfectly, exactly matching its semantics (to the extent that C has semantics), and still fail to deliver the behavior needed by the system. For example, it could miss timing deadlines. Since timing is not in the semantics of C, whether a program misses deadlines is in fact irrelevant to determining whether it has executed correctly. But it is very relevant to determining whether the *system* has performed correctly. A component that is perfectly predictable and repeatable turns out not to be predictable and repeatable in the dimensions that matter. This is a failure of abstraction.

The problem gets worse as software systems get more complex. If we step outside C and use operating system primitives to perform I/O or to set up concurrent threads, we immediately move from essentially perfect predictability and repeatability to wildly nondeterministic behavior that must be carefully reigned in by the software designer [18]. Semaphores, mutual exclusion locks, transactions, and priorities are some of the tools

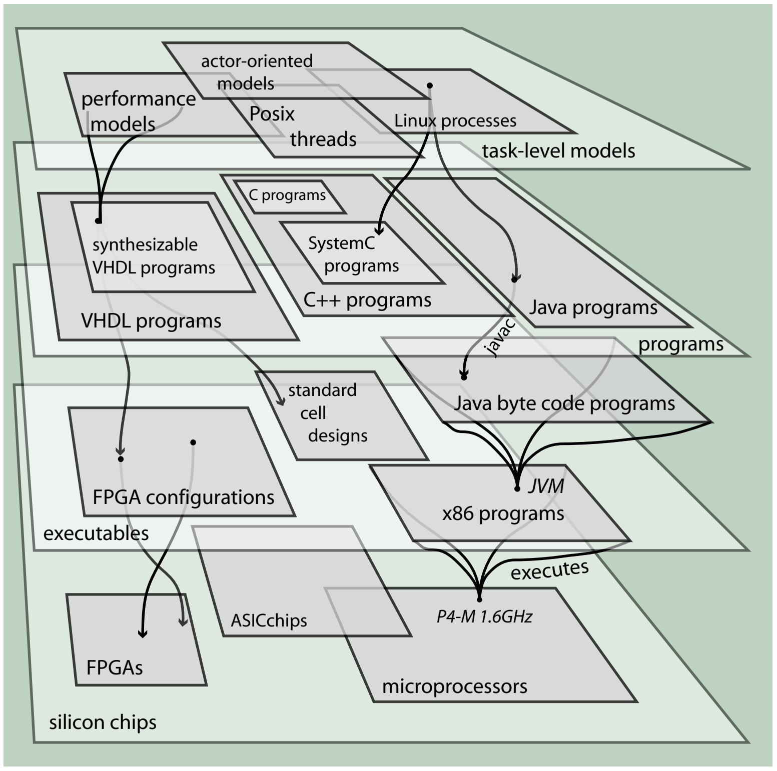

Figure 1: Abstraction layers in computing.

that software designers have developed to attempt to compensate for this loss of predictability and repeatability.

But the question we must ask is whether this loss of predictability and repeatability is really necessary. I believe it is not. If we find a way to deliver predictable software with repeatable behavior (with respect to behavior properties that matter, such as timing), then we do not eliminate the need to design robust systems, but we dramatically change the nature of the challenge. We must follow the principle of making systems predictable and repeatable if this is technically feasible, and give up only when there is convincing evidence that this is not possible or cost effective. There is no such evidence for software. Moreover, we have an enormous asset: the substrate on which we build software systems (digital circuits) is essentially perfectly predictable and repeatable with respect to properties we care about (timing and logical functionality).

Let us examine further the failure of abstraction. 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 of designs. E.g., 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 design. Above that is the set of all x86 programs, each of which can run on that processor. This set is defined precisely (unlike the previous set, which is difficult to define) by the x86 instruction set architecture (ISA). Any program coded in that instruction set is a member of the set; for example, a particular implementation of a Java virtual machine may be a member of the set. Associated with that member is another set, the set of all JVM bytecode programs. Each of these programs is (typically) synthesized by a compiler from a Java program, which is a member of the set of all syntactically valid Java programs. Again, this set is defined precisely by 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 current state of embedded software, 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 cannot express. 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 program from their concurrent orchestration, yet this fails if the timing of the underlying platform is not repeatable, or if execution times cannot be determined. The network hides details of electrical or optical signaling from systems, but most standard networks provide no timing guarantees and fail to provide an appropriate abstraction. A system designer is stuck with a system *design* (not just implementation) in silicon and wires.

All embedded systems designers face versions of this problem. Aircraft manufacturers have to stockpile the electronic parts needed for the entire production line of an aircraft model to avoid having to recertify the software if the hardware changes. “Upgrading” a microprocessor in an engine control unit for a car requires thorough re-testing of the system. Even “bug fixes” in the software or hardware can be extremely risky, since they can change timing behavior.

The design of an abstraction layer involves many choices, and computer scientists have chosen to hide timing properties from all higher abstractions. Wirth [38] 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 an embedded system, however, computations interact directly with the physical world, where time cannot be abstracted away. Even general-purpose computing suffers from these choices. Since timing is neither specified in programs nor enforced by execution platforms, a program's timing is not repeatable. Concurrent software often has timing-dependent behavior in which small changes in timing have big consequences, introducing a new form of brittle behavior. And the vast number of possible interleavings of threads makes predictability intractable.

Designers have traditionally covered these failures by finding worst case execution time (WCET) bounds [37] and using real-time operating systems (RTOS's) with well-understood scheduling policies [8]. Despite recent improvements, these often require substantial margins for reliability, particularly as processor architectures develop ever more elaborate techniques for dealing stochastically with deep pipelines, memory hierarchy, and parallelism [36, 14]. Modern processor architectures render WCET virtually unknowable; even simple problems demand heroic efforts. In practice, reliable WCET numbers come with many caveats that are increasingly rare in software. Worse, any analysis that is done, no matter how tight the bounds are, applies to only a very specific program on a very specific piece of hardware. Any change in either the hardware or software, no matter how small, renders the analysis invalid. The processor ISA has failed to provide an adequate abstraction. Still worse, even perfectly tight WCET bounds for software components does not guarantee repeatability. The so-called "Richard's anomalies," explained nicely in [8], show that under very popular EDF scheduling policies, the mere fact that all tasks finish *early* can cause deadlines to be missed that would not have been missed if the tasks had finished at the WCET bound. Designers have to be very careful to analyze their scheduling strategies under worst case *and* best case execution times, and everything in between.

Timing behavior in RTOSs 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 often fails to identify subtle timing bugs. Worse, these techniques produce brittle systems in which small changes can cause big failures. As a telling example, Patrick Lardieri of Lockheed Martin discussed some experiences with the Joint Strike Fighter Program, saying[4] "Changing the instruction memory layout of the Flight Control Systems Control Law process to optimize Built in Test processing led to an unexpected performance change - [the] System went from meeting real-time requirements to missing most deadlines due to a change that was expected to have no impact on system performance."

While there are no true guarantees in life, we should not blithely discard predictability and repeatability that *is* achievable. Synchronous digital hardware—the technology on which computers are built— delivers astonishingly precise timing behavior reliably. Software abstractions, however, discard several orders of magnitude of precision. Compare the nanosecond-scale precision with which hardware can raise an interrupt request to the millisecond-level precision with which software threads respond. We don't have to do it this way.

## 5 Solutions

The problems raised in this paper pervade computing abstractions from top to bottom. As a consequence, most specialities within computer science have work to do. This section suggests a few directions. All of these draw on current and past contributions, thus suggesting that the vision outlined in this paper, albeit radical, is indeed achievable. We do not need to restart from scratch.

### 5.1 Computer Architecture

The ISA of a processor provides an abstraction of computing hardware for the benefit of software designers. The value of this abstraction is enormous. Among the benefits is that generations of CPUs that implement the same ISA can have different performance without compromising compatibility with existing software. Today's ISAs hide most temporal properties of the underlying hardware. Perhaps the time is right to augment the ISA abstraction with carefully selected timing properties, so that this compatibility extends as well to time-sensitive systems.

In 1980, Patterson and Ditzel [31] argued that computer architects had gone overboard with specialized and complex instructions in the instruction set. They argued for a back-to-basics approach to architecture, launching

[4]National Workshop on High-Confidence Software Platforms for Cyber-Physical Systems (HCSP-CPS), Arlington, VA November 30 December 1, 2006.

the era of RISC machines. Perhaps a similar retrenchment is needed today, but this time to recover predictable and repeatable timing with a new generation of "precision timed" (PRET) machines [10].

Of course, achieving timing precision is easy if we are willing to forgo performance; the engineering challenge is to deliver both precision and performance. For example, although cache memories may introduce unacceptable timing variability, we cannot do without memory hierarchy. The challenge is to provide memory hierarchy with repeatable behavior. Similar challenges apply to pipelining, bus architectures, and I/O mechanisms. Some progress in this direction is reported in [23].

### 5.2 Programming Languages

Programming languages provide an abstraction layer above the ISA. If the ISA is to expose selected temporal properties, and programmers wish to exploit this, then one approach would be to reflect these in the languages.

There is a long and somewhat checkered history of attempts to insert timing features into programming languages. Ada can express a delay operation, but not timing constraints. Real-Time Java augments the Java model with a few ad-hoc features that reduce variability of timing [7]. The synchronous languages [5], such as Esterel, Lustre, and Signal, do not have explicit timing constructs in them, but because of their predictable and repeatable approach to concurrency, can yield more predictable and repeatable timing than most alternatives. They are only limited by the underlying platform. Much earlier, Modula-2 [39] gives control over scheduling of co-routines, which makes it possible, albeit laborious, for programmers to exercise some coarse control over timing. Like the synchronous languages, timing properties of the program are not explicit in the program. Real-time Euclid [15], on the other hand, expresses process periods and absolute start times.

Rather than new languages, an alternative is to annotate programs written in conventional languages. Lee [21] gives a taxonomy of timing properties that must be expressible in such annotations. Münzenberger et al. [29] give annotations for SDL to express real-time constraints. TimeC [22] introduces extensions to specify timing requirements based on events, with the objective of controlling code generation in compilers to exploit instruction level pipelining.

Domain-specific languages with temporal semantics have firmly taken hold in some areas. Simulink, from The MathWorks, provides a graphical syntax and language for timed systems that can be compiled into embedded real-time code for control systems. LabVIEW, from National Instruments, recently added timed extensions. It is widely used in instrumentation systems. A much earlier example is PEARL [28], also aimed at control systems; PEARL could specify absolute and relative start times, deadlines, and periods, and was fairly widely used at the time.

All of these, however, remain outside the mainstream of software engineering. They are not well integrated into software engineering processes and tools, and they have not benefited from many innovations in programming languages.

## 5.3 Software Component Technologies

Software engineering innovations such as data abstraction, object-orientation, and component libraries have made it much easier to design large complex software systems. Today's most successful component technologies (class libraries and utility functions) do not export even the most rudimentary temporal properties in their APIs. Although a knowledgeable programmer may be savvy enough to use a hash table over a linked list when random access is required, the API for these data structures expresses nothing about access times. Component technologies with temporal properties will be required, and in fact provide an attractive alternative to real-time programming languages.

An early example, Larch [4], gives a task-level specification language integrating functional descriptions with timing constraints. Other examples function at the level of *coordination languages* rather than specification languages. A coordination language executes at run time, whereas a specification language does not. For example, Broy [6] focuses on timed concurrent components communicating via timed streams. Zhao et al. [40] give an actor-based coordination language for distributed real-time systems based on discrete-event systems semantics. New coordination languages where the components are given using established programming languages (such as Java and C++) may be more likely to gain acceptance than new programming languages that replace the established languages. When coordination languages acquire rigorous timed semantics, designs function more like *models* than *programs* [13].

But many challenges remain in developing this relatively immature technology. Naive abstractions of time, such as the discrete-time models commonly used to analyze control and signal processing systems, do not reflect the true behavior of software and networks [30]. The concept of "logical execution time" [11] offers a more promising abstraction, but ultimately still relies on being able to get worst-case execution times for software components. This top-down solution depends on a corresponding bottom-up solution.

## 5.4 Formal Methods

Formal methods use mathematical models to infer and prove properties of systems. Formal methods that handle temporal dynamics are less prevalent than those that handle sequences of state changes, but there is good work on which to draw. For example, in *interface theories* [9], software components export temporal interfaces, and behavioral type systems validate the composition of components and infer interfaces for compositions of components. Specific interface theories of this type are given in [35, 17].

Various temporal logics support reasoning about timing properties of systems [12, 3]. Temporal logics mostly deal with "eventually" and "always" properties to reason about safety and liveness, but various extensions support metric time [1, 27, 2]. A few process algebras also support reasoning about time (see for example [32, 25, 20]). The most accepted formalism for the specification of real-time requirements is timed automata (and variations thereof) [2].

Another approach uses static analysis of programs coupled with models of the underlying hardware [37]. This approach is gaining traction in industry, but suffers from some fundamental limitations. The most important one is brittleness. Even very small changes in either the hardware or the software invalidate the analysis. A less important limitation, nonetheless worth noting, is that the use of Turing-complete programming languages and models leads to undecidability. Not all programs can be analyzed.

All of these techniques enable some form of formal verification. However, properties that are not formally specified cannot be formally verified. Thus, for example, timing behavior of software that is not expressed in the software, must be separately specified, and the connection between specifications and between specification and implementations becomes tenuous. This solution depends on progress in programming languages. Moreover, despite considerable progress in automated abstraction, scalability to realistic systems remains a major issue. Although offering a wealth of elegant results, the impact of most of these formal techniques on engineering practice has been small (not zero, but small). In general-purpose computing, type systems are formal methods that have had enormous impact. What is needed is *time systems* with the power of type systems.

## 5.5 Operating Systems

One of the key services of an operating system is scheduling. Scheduling of real-time tasks, of course, is a venerable, established area of inquiry. Classic techniques like rate-monotonic scheduling (RMS) and earliest deadline first (EDF) are well studied and have many elaborations. With a few exceptions [16, 11], the field has seen less emphasis on *repeatability* over optimization. Consider a concrete challenge: to get repeatable real-time behavior, a CPS designer may use the notion of logical execution time (LET) [11] for the time-sensitive portions of a system, and best-effort execution for the less time-sensitive portions. The best-effort portions will typically not have deadlines, and hence EDF will give them lowest priority. However, the correct optimization is to execute the best-effort portions as early as possible subject to the constraint that the LET portions match their timing specifications. Even though the LET portions have deadlines, they should not necessarily get higher priority than the best effort portions.

Today, embedded system designers avoid mixing time-sensitive operations with best effort ones. Every cell phone currently in use has at least two CPUs in it, one for the hard real-time tasks of speech coding and radio functions, and one for the user interface, database, email, and networking functionality. The situation is worse in cars and manufacturing systems, where distinct CPUs tend to be used for a myriad of distinct features. The design is this way not because there are not enough cycles in today's CPUs to combine the tasks, but rather because we do not have reliable technology for mixing distinct types of tasks. My opinion is that a focus on repeatability of timing behavior could lead to such a technology. Work on deferrable/sporadic servers [24] may provide a promising point of departure.

## 5.6 Networking

In the context of general-purpose networks, timing behavior is viewed as a quality of service (QoS) problem. Considerable activity a decade or two ago led to many ideas for addressing QoS concerns, few of which were deployed with any impact. Today, designers of time sensitive applications on general-purpose networks, such as voice over IP (VOIP), struggle with inadequate control over network behavior.

Meanwhile, in the embedded systems space, specialized networks such as FlexRay and the time-triggered architecture (TTA) [16] emerged to provide timing as a correctness property rather than a QoS property. A flurry of recent activity has led to a number of innovations such as time synchronization (IEEE 1588), synchronous ethernet, time-triggered ethernet, etc. At least one of these (synchronous ethernet) is encroaching on general-purpose networking, driven by the demand for convergence of telephony and video services with the internet, as well as by the potential for real-time interactive games. My opinion is that introducing timing into networks as a semantic property rather than a QoS problem will lead to an explosion of new time-sensitive applications, helping to realize the vision of CPS.

## 6 Conclusion

To fully realize the potential of CPS, the core abstractions of computing need to be rethought. Incremental improvements will, of course, continue to help. But effective orchestration of software and physical processes requires semantic models that reflect properties of interest in both.

This paper has focused on making temporal dynamics explicit in computing abstractions, so that timing properties become correctness criteria rather than quality of service measures. I have argued for making timing of programs and networks as repeatable and predictable as is technologically feasible at reasonable cost. This will not eliminate timing variability, and hence does not eliminate the need for adaptive techniques and validation methods that work with bounds on timing. But it does eliminate spurious sources of timing variability, and enables precise and repeatable timing when this is needed. The result will be computing and networking technologies that enable vastly more sophisticated cyber-physical systems.

## 7 Acknowledgments

Special thanks to Tom Henzinger, Insup Lee, Al Mok, Sanjit Seshia, Jack Stankovic, Lothar Thiele, Reinhard Wilhelm, Moshe Vardi, and anonymous reviewers for very helpful comments and suggestions.

## References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1543 – 1571, 1994.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] R. Alur and T. Henzinger. Logics and models of real time: A survey. In J. W. De Bakker, C. Huizing, W. P. De Roever, and G. Rozenberg, editors, *REX Workshop*, volume LNCS 600, pages 74–106, Mook, The Netherlands, June 3-7 1991. Springer.

[4] M. R. Barbacci and J. M. Wing. Specifying functional and timing behavior for real-time applications. Technical Report ESD-TR-86-208, Carnegie Mellon University, December 1986.

[5] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.

[6] M. Broy. Refinement of time. *Theoretical Computer Science*, 253:3–26, 2001.

[7] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada 95, Real-Time Java and Real-Time POSIX.* Addison-Wesley, 3d edition, 2001.

[8] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications.* Springer, second edition, 2005.

[9] L. deAlfaro and T. A. Henzinger. Interface theories for component-based design. In *First International Workshop on Embedded Software (EMSOFT)*, volume LNCS 2211, pages 148–165, Lake Tahoe, CA, October, 2001 2001. Springer-Verlag.

[10] S. A. Edwards and E. A. Lee. The case for the precision timed (PRET) machine. In *Design Automation Conference (DAC)*, San Diego, CA, June 4-8 2007.

[11] 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.

[12] F. Jahanian and A. K. Mok. Safety analysis of timing properties in real-time systems. *IEEE Tr. on Software Engineering*, 12(9), 1986.

[13] G. Karsai, J. Sztipanovits, A. Ledeczi, and T. Bapty. Model-integrated development of embedded software. *Proceedings of the IEEE*, 91(1):145–164, 2003.

[14] R. Kirner and P. Puschner. Obstacles in worst-case execution time analysis. In *Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 333–339, Orlando, FL, USA, May 5-7 2008. IEEE.

[15] E. Klingerman and A. Stoyenko. Real-time Euclid: A language for reliable real-time systems. *IEEE Tr. on Software Engineering*, 12(9), 1986.

[16] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[17] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2003)*, pages 51–60, Hakodate, Hokkaido, Japan, 14-16 May 2003 2003. IEEE Computer Society.

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

[19] E. A. Lee and A. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Circuits and Systems*, 17(12):1217–1229, 1998.

[20] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, pages 158–171, 1994. develops a resource-based (real-time) process algebra.

[21] I. Lee, S. Davidson, and V. Wolfe. Motivating time as a first class entity. Technical Report MS-CIS-87-54, Dept. of Comp. and Infor. Science, Univ. of Penn, Aug. (Revised Oct.) 1987.

[22] A. Leung, K. V. Palem, and A. Pnueli. TimeC: A time constraint language for ILP processor compilation. Technical Report TR1998-764, New York University, 1998.

[23] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, Atlanta, October 2008.

[24] J. W. S. Liu. *Real-Time Systems*. Prentice-Hall, 2000. According to Insup Lee, has a good description of deferrable/sporadic servers.

[25] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. Technical Report EECS-2006-67, to appear in *Theoretical Computer Science*, UC Berkeley, May 18 2006.

[26] O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In *Real-Time: Theory and Practice, REX Workshop*, pages 447–484. Springer-Verlag, 1992.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, Berlin, 1992.

[28] T. Martin. Real-time programing language PEARL - concept and characteristics. In *COMPSAC*, pages 301–306, Chicago, 1978.

[29] R. Münzenberger, M. Drfel, R. Hofmann, and F. Slomka. A general time model for the specification and design of embedded real-time systems. *Microelectronics Journal*, 34:989–1000, 2003.

[30] T. Nghiem, G. J. Pappas, A. Girard, and R. Alur. Time-triggered implementations of dynamic controllers. In *EMSOFT*, pages 2–11, Seoul, Korea, 2006. ACM Press.

[31] 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.

[32] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.

[33] J. A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, 1988.

[34] J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar. Opportunities and obligations for physical computing systems. *Computer*, pages 23–31, 2005.

[35] L. Thiele, E. Wandeler, and N. Stoimenov. Real-time interfaces for composing real-time systems. In *EMSOFT*, Seoul, Korea, October 23-25 2006. ACM Press.

[36] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[37] 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. Stenstr. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

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

[39] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.

[40] Y. Zhao, E. A. Lee, and J. Liu. A programming model for time-synchronized distributed real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Bellevue, WA, USA, April 3-6 2007. IEEE.