Cyber-Physical Systems - Are Computing Foundations Adequate?

Edward A. Lee

Department of EECS, UC Berkeley

Position Paper for
NSF Workshop On Cyber-Physical Systems:
Research Motivation, Techniques and Roadmap
October 16 - 17, 2006
Austin, TX

1 Summary

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

I argue that the mismatch between these abstractions and properties of physical processes impede technical progress, and I identify promising technologies for research and investment. There are technical approaches that partially bridge the abstraction gap today (such as real-time operating systems, middleware technologies, specialized embedded processor architectures, and specialized networks), and there is certainly considerable room for improvement of these technologies. However, it may be that we need a less incremental approach, where new abstractions are built from the ground up.

The foundations of computing are built on the premise that the principal task of computers is transformation of data. Yet we know that the technology is capable of far richer interactions the physical world. I critically examine the foundations that have been built over the last several decades, and determine where the technology and theory bottlenecks and opportunities lie. I argue for a new systems science that is jointly physical and computational.

2 Applications of CPS

Applications of CPS arguably have the potential to dwarf the 20-th century IT revolution. They include high confidence medical devices and systems, 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, such as distributed micro power generation coupled into the power grid, where timing precision and security issues loom large. Transportation systems could benefit considerably from better embedded intelligence in automobiles, which could improve safety and efficiency. Networked autonomous vehicles could dramatically enhance the effectiveness of our military and could offer substantially more effective disaster recovery techniques. Networked building control systems (such as HVAC and lighting) could significantly improve energy efficiency and demand variability, reducing our dependence on fossil fuels. In communications, cognitive radio could benefit enormously from distributed consensus about available bandwidth and from distributed control technologies. Financial networks could be dramatically changed by precision timing. Large scale services systems leveraging RFID and other technologies for tracking of goods and services could acquire the nature of distributed real-time control systems. Distributed real-time games that integrate sensors and actuators could change the (relatively passive) nature of on-line social interactions. Tight integration of physical devices and distributed computing could make "programmable matter" a reality.

The positive economic impact of any one of these applications areas would be enormous. Today's computing and networking technologies, however, may have properties that unnecessarily impede progress towards these applications. For example, the lack of temporal semantics and adequate concurrency models in computing, and today's "best effort" networking technologies make predictable and reliable real-time performance difficult, at best. Many of these applications may not be achievable without substantial changes in the core abstractions.

3 Technologies

Integration of physical processes and computing, of course, is not new. The term "embedded systems" has been used for some time to describe engineered systems that combine physical processes with computing. Successful applications include communication systems, aircraft control systems, automotive electronics, home appliances, weapons systems, games and toys, etc. However, most such embedded systems are closed "boxes" that do not expose the computing capability to the outside. The radical transformation that we envision comes from networking these devices.

Such networking poses considerable technical challenges. For example, prevailing practice in embedded software relies on bench testing for concurrency and timing properties. This has worked reasonably well, because programs are small, and because software gets encased in a box with no outside connectivity that can alter the behavior. However, the applications we envision demand that embedded systems be feature-rich and networked, so bench testing and encasing become inadequate. In a networked environment, it becomes impossible to test the software under all possible conditions. Moreover, general-purpose network-

ing techniques themselves make program behavior much more unpredictable. A major technical challenge is to achieve predictable timing in the face of such openness.

Before DARPA began investing in embedded systems in the mid-1990s (principally through the MoBIES, SEC, and NEST programs), the research community devoted to this problem was small. Embedded systems were largely an industrial problem, one of using small computers to enhance the performance or functionality of a product. In this earlier context, embedded software differed from other software only in its resource limitations (small memory, small data word sizes, and relatively slow clocks). In this view, the "embedded software problem" is an optimization problem. Solutions emphasize efficiency; engineers write software at a very low level (in assembly code or C), avoid operating systems with a rich suite of services, and use specialized computer architectures such as programmable DSPs and network processors that provide hardware support for common operations. These solutions have defined the practice of embedded software design and development for the last 25 years or so. In an analysis that remains as valid today as 18 years ago, Stankovic [18] laments the resulting misconceptions that real-time computing "is equivalent to fast computing" or "is performance engineering" (most embedded computing is real-time computing).

But the resource limitations of 25 years ago are surely not resource limitations today. Indeed, the technical challenges have centered more on predictability and robustness than on efficiency. Safety-critical embedded systems, such as avionics control systems for passenger aircraft, are forced into an extreme form of the "encased box" mentality. For example, in order to assure a 50 year production cycle for a fly-by-wire aircraft, an aircraft manufacturer is forced to purchase, all at once, a 50 year supply of the microprocessors that will run the embedded software. To ensure that validated real-time performance is maintained, these microprocessors must all be manufactured on the same production line from the same masks. The systems will be unable to benefit from the next 50 years of technology improvements without redoing the (extremely expensive) validation and certification of the software. Evidently, efficiency is nearly irrelevant compared to predictability, and predictability is difficult to achieve without freezing the design at the physical level. Clearly, something is wrong with the software abstractions being used. Although raising the level of abstraction can help [19], I will argue that the core abstractions need to be rethought.

A notable possible culprit is the lack of timing in computing abstractions. Indeed, this lack has been exploited heavily in such computer science disciplines as architecture, programming languages, operating systems, and networking. In architecture, for example, although synchronous digital logic delivers precise timing determinacy, advances have made it difficult or impossible to estimate or predict the execution time of software. Modern processor architectures use memory hierarchy (caches), dynamic dispatch, and speculative execution to improve average case performance of software, at the expense of predictability. These techniques make it nearly impossible to tell how long it will take to execute a particular

piece of code.[1] To deal with these architectural problems, embedded software designers may choose alternative processor architectures such as programmable DSPs not only for efficiency reasons, but also for predictability of timing. This study will examine the circumstances under which such choices are made and identify the trends in architecture that might benefit the most timing-sensitive CPS applications.

Even less timing-sensitive applications have been affected. Anecdotal information from computer-based instrumentation, for example, indicates that the real-time performance delivered by today's PCs is about the same as was delivered by PCs in the mid-1980's. Twenty years of Moore's law have not improved things in this dimension. This is not entirely due to hardware architecture techniques, of course. Operating systems, programming languages, user interfaces, and networking technologies have become more elaborate. All have been built on an abstraction of software where time is irrelevant. No widely used programming language includes temporal properties in its semantics, and "correct" execution of a program has nothing to do with time. Benchmarks emphasize average-case performance, and timing predictability is irrelevant.

The prevailing view of real-time appears to have been established well before embedded computing was common. Wirth [22] reduces real-time programming to threads with bounds on execution time, arguing that "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 this sequential framework, "computation" is accomplished by a terminating sequence of state transformations. This core abstraction underlies the design of nearly all computers, programming languages, and operating systems in use today. But unfortunately, this core abstraction may not fit CPS very well.

The most interesting and revolutionary cyber-physical systems will be networked. The most widely used networking techniques today introduce a great deal of timing variability and stochastic behavior. Today, embedded systems are often forced to use less widely accepted networking technologies (such as CAN busses in manufacturing systems and FlexRay in automotive applications), and typically must limit the geographic extent of these networks to a confined local area. What aspects of those networking technologies should or could be important in larger scale networks? Which are compatible with global networking techniques?

To be specific, recent advances in time synchronization across networks promise networked platforms that share a common notion of time to a known precision [11]. How would that change how distributed cyber-physical applications are developed? What are the implications for security? Can we mitigate security risks created by the possibility of disrupting the shared notion of time? Can security

[1] A glib response is that execution time in a Turing-complete language is undecidable anyway, so it's not worth even trying to predict execution time. This is nonsense. No cyber-physical system that depends on timeliness can be deployed without timing assurances. If Turing completeness interferes with this, then Turing completeness must be sacrificed.

techniques effectively exploit a shared notion of time to improve robustness? In particular, although distributed denial of service attacks have proved surprisingly difficult to contend with in general purpose IT networks, could they be controlled in time synchronized networks?

Operating systems technology is also groaning under the weight of the requirements of embedded systems. So-called "real-time operating systems" (RTOS's), are still essentially best-effort technologies. Are they merely a temporary patch for inadequate computing foundations? What would replace them? Is the conceptual boundary between the operating system and the programming language (a boundary established in the 1960's) still the right one? It would be truly amazing if it were.

Cyber-physical systems by nature will be concurrent. Physical processes are intrinsically concurrent, and their coupling with computing requires, at a minimum, concurrent composition of the computing processes with the physical ones. Even today, embedded systems must react to multiple real-time streams of sensor stimuli and control multiple actuators concurrently. Regrettably, the mechanisms of interaction with sensor and actuator hardware, built for example on the concept of interrupts, are not well represented in programming languages. They have been deemed to be the domain of operating systems, not of software design. Instead, the concurrent interactions with hardware are exposed to programmers through the abstraction of threads.

Threads, however, are a notoriously problematic [14, 23]. This fact is often blamed on humans rather than on the abstraction. Sutter and Larus [20] observe that "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations." The problem will get far worse with extensively networked cyber-physical systems.

Yet humans are actually quite adept at reasoning about concurrent systems. The physical world is highly concurrent, and our very survival depends on our ability to reason about concurrent physical dynamics. The problem is that we have chosen concurrent abstractions for software that do not even vaguely resemble the concurrency of the physical world. We have become so used to these computational abstractions that we have lost track of the fact that they are not immutable. Could it be that the difficulty of concurrent programming is a consequence of the abstractions, and that if we were are willing to let go of those abstractions, then the problem would be fixable?

Embedded computing also exploits concurrency models other than threads. Programmable DSP architectures are often VLIW machines. Video signal processors often combine SIMD with VLIW and stream processing. Network processors provide explicit hardware support for streaming data. However, despite considerable innovative research, in practice, programming models for these domains remain primitive. Designers write low-level assembly code that exploits specific hardware features, and combine this assembly code with C code only where performance is not so critical.

For the next generation of cyber-physical systems, it is arguable that we must build concurrent models of computation that are far more deterministic, predictable, and understandable. Threads take the opposite approach. They make programs absurdly nondeterministic, and rely on programming style to constrain that nondeterminism to achieve deterministic aims. Can a more deterministic approach be reconciled with the intrinsic need for nondeterminism in many embedded applications? How should cyber-physical systems contend with the inherent unpredictability of the (networked) physical world?

4 Research Directions

Fortunately, there is a great deal of work on which to build the CPS research agenda. A sampling of promising directions is given below.

- *Putting time into programming languages.* Since the mid 1980's, a number of experimental languages have been developed with temporal semantics. The most successful of these have been domain-specific, including both modeling languages such as Simulink from the MathWorks and coordination languages such as Giotto [9].
- *Rethinking the OS/programming language split.* Operating systems as a concept originated in the 1960s when the landscape of computing was very different. The particular ways in which they abstract the hardware and isolate the programming languages from hardware-dependent behaviors needs to be rethought for CPS. A particularly promising start is TinyOS/nesC [10], where the programming language abstraction in nesC supports the design of thin wrappers around hardware. Moreover, the concurrency model in nesC mirrors that of interrupt-driven hardware interactions that are so common in embedded systems. TinyOS and nesC together blur the boundary between the operating system and the programming language, and offer a more suitable set of abstractions for embedded software design.
- *Rethink the hardware/software split.* Cyber-physical systems intrinsically combine hardware and physical systems with software. Today, the modeling and design abstractions used for hardware and physical systems are entirely different from those used for software, and few modeling or design languages support mixtures of the two. A promising start is the electronic design automation (EDA) community's explorations with hardware/software codesign. Model-based design efforts that focus on heterogeneous mixtures of models of computation, such as the Ptolemy Project [7] also have ideas to offer.
- *Memory hierarchy with predictability.* Memory hierarchy techniques have had a major impact on the scaling of computing performance. However, the techniques that are most widespread deliver improved performance and memory capacity at the expense of timing predictability. This need not be so. For example, scratchpad memories shift the control of the memory hierarchy from hardware to software [1]. Combined with languages that have temporal

semantics, this technique could deliver timing guarantees through compiler techniques.

- *Memory management with predictability.* Managing allocation and deallocation of memory manually is error-prone and tedious. Automatic memory management significantly improves programmer productivity and software reliability, but again at the expense of timing predictability. A promising start is a family of techniques that deliver bounded pause time garbage collection [2, 17].
- *Predictable, controllable deep pipelines.* Deep pipelining is essential for competitive performance in computing. However, dependencies between instructions means that the rate at which instructions are executed must be modulated to ensure that data dependencies are met. Today's techniques, such as dynamic dispatch and speculative execution, again deliver high performance at the expense of timing predictability. There are, however, alternatives. For example, pipeline interleaving converts pipeline latencies into parallelism and enables full utilization of the pipeline with predictable timing [15]. Combining this method with stream-oriented programming exposes sufficient parallelism for many CPS-style applications [16]. A recent variant of pipeline interleaving called hyper-threading is disappointing because it does little for timing predictability.
- *Predictable, controllable, understandable concurrency.* Cyber-physical systems are intrinsically concurrent. Today, most widely used concurrent programming techniques, such as threads, deliver incomprehensible and unreliable code [14]. More predictable, controllable, and understandable techniques for dealing with concurrency have been developed. For example, the synchronous languages [3] have found effective use in safety-critical avionics systems because of their of their more manageable concurrency model [4].
- *Concurrent components.* Component software technologies, most particularly object-oriented design, have delivered enormous improvements in the design of large software systems. However, these technologies are deeply rooted in sequential, imperative computational models. Hence, they do not adapt well to concurrent real-time computing. Promising alternatives are actor-oriented component models [13] and model-based design techniques [21].
- *Networks with timing.* Today's general purpose networking techniques, such as TCP/IP, are best-effort techniques over which it is very difficult to achieve timing predictability. More specialized networking techniques, however, have been developed and have found effective use in embedded systems [12]. Moreover, network time synchronization technology has improved considerably, offering the possibility of timing coherence across distributed computations [11]. In combination, these two networking innovations could dramatically change the way distributed real-time software is designed.
- *Computational dynamical systems theory.* Systems theories today are either purely physical (e.g. control systems, signal processing) or purely computational (e.g. computability, complexity). A few blended theories have emerged,

including hybrid systems [8] and timed process algebras [6]. Moreover, instead of complexity, one could analyze schedulability [5].

The foundational challenges of CPS require top-to-bottom rethinking of computation. I identify properties in the core abstractions of computing that are obstacles to progress in cyber-physical systems, and also identify promising research directions and technologies on which can build. The resulting research agenda is enormous, spanning most disciplines within computer science and several within other engineering disciplines such as mechanical and electrical.

References

1. O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Trans. on Embedded Computing Sys.*, 1(1):6–26, 2002.
2. D. F. Bacon, P. Cheng, and V. Rajan. The Metronome: A simpler approach to garbage collection in real-time systems. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, Catania, Sicily, November 2003.
3. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, September 1991.
4. G. Berry. The effectiveness of synchronous languages for the development of safety-critical systems. White paper, Esterel Technologies, 2003.
5. E. Bini and G. C. Buttazzo. Schedulability analysis of periodic fixed priority systems. *IEEE Transactions on Computers*, 53(11):1462–1473, November 2004.
6. M. Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology*, 2(1):1–46, January 1993.
7. J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(2):127–144, January 2003.
8. T. A. Henzinger. The theory of hybrid automata. In M. Inan and R. Kurshan, editors, *Verification of Digital and Hybrid Systems*, volume 170 of *NATO ASI Series F: Computer and Systems Sciences*, pages 265–292. Springer-Verlag, 2000.
9. 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.
10. J. Hill, R. Szewcyk, A. Woo, D. Culler, S. Hollar, and K. Pister. System architecture directions for networked sensors. In *9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, November 2000.
11. S. Johannessen. Time synchronization in a local area network. *IEEE Control Systems Magazine*, pages 61–69, April 2004.
12. H. Kopetz and G. Grunsteidl. TTP - a protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, January 1994 1994.
13. E. A. Lee. Model-driven development - from object-oriented design to actor-oriented design. In *Workshop on Software Engineering for Embedded Systems: From Requirements to Implementation (a.k.a. The Monterey Workshop)*, Chicago, September 24 2003.
14. E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

15. E. A. Lee and D. G. Messerschmitt. Pipeline interleaved programmable dsps: Architecture. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-35(9), September 1987.
16. E. A. Lee and D. G. Messerschmitt. Pipeline interleaved programmable dsps: Synchronous data flow programming. *IEEE Trans. on Acoustics, Speech, and Signal Processing*, ASSP-35(9), September 1987.
17. T. Ritzau and P. Fritzson. Decreasing memory overhead in hard real-time garbage collection. In *EMSOFT*, Grenoble, October, 2002 2002. Springer-Verlag.
18. J. A. Stankovic. Misconceptions about real-time computing: a serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988.
19. J. A. Stankovic, I. Lee, A. Mok, and R. Rajkumar. Opportunities and obligations for physical computing systems. *Computer*, pages 23–31, November 2005.
20. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, September 2005.
21. J. Sztipanovits and G. Karsai. Model-integrated computing. *IEEE Computer*, page 110112, April 1997 1997.
22. N. Wirth. Toward a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.
23. N. Zeldovich, A. Yip, F. Dabek, R. T. Morris, D. Mazieres, and F. Kaashoek. Multiprocessor support for event-driven programs. In *USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 9-14 2003.