Absolutely Positively on Time: What Would It Take?

Edward A. Lee, University of California, Berkeley

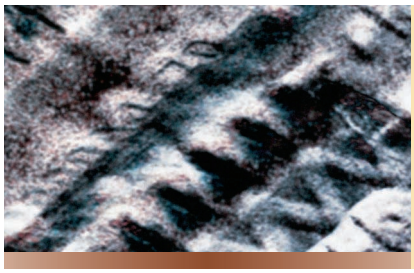
Despite considerable progress in software and hardware techniques, many recent computing advances do more harm than good when embedded computing systems absolutely must meet tight timing constraints.

For example, while synchronous digital logic delivers precise timing determinacy, advances in computer architecture and software have made it difficult or impossible to estimate or predict software's execution time. Moreover, networking techniques introduce variability and stochastic behavior, while operating systems rely on best-effort techniques. Worse, programming language semantics do not handle time well, so developers can only specify timing requirements indirectly.

Thus, achieving precise timeliness in a networked embedded system—an absolutely essential goal—will require sweeping changes.

CORE ABSTRACTION

Contemporary computer science has taught us that a Turing machine can specify everything that can be computed. Computation is accomplished by a terminating sequence of state transformations. A *computable function* provides a map from a bit sequence to a bit sequence. This core abstraction underlies the design of most computers, programming languages, and operating systems currently in use.



For embedded computing to realize its full potential, we must reinvent computer science.

Unfortunately, it does not fit embedded software well. If, however, "embedded software" is simply "software on small computers," then this abstraction fits *reasonably* well. In this view, embedded software differs 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 one of optimization.

Optimizing 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 past 25 years or so.

MUCH PROGRESS, LITTLE CHANGE

Given the semiconductor industry's ability to keep pace with Moore's law, the resource limitations of 25 years ago should have almost entirely evaporated by now. Yet embedded software design and development have changed little.

This lack of change may stem from the extreme competitive pressure in products such as consumer electronics, which are based on embedded software and reward only the most efficient solutions. There are, however, many examples where functionality and reliability have proven more important than efficiency, which makes it arguable that factors other than—and possibly even as important as—resource limitations have influenced embedded software's evolution.

Embedded software differs from other software in more fundamental ways. Examining why engineers write embedded software in assembly code or C reveals that efficiency is not their only concern, and may not even be their main one. Reasons for this could include the need to count cycles in a critical inner loop—not to make it fast, but rather to make it predictable.

No widely used programming language integrates a way to specify timing requirements or constraints. Instead, the abstractions they offer focus on scalability—inheritance, dynamic binding, polymorphism, memory management—and, if anything, further obscure timing. Consider, for example, the impact of garbage collection on timing.

Counting cycles becomes extremely difficult on modern processor architectures, where memory hierarchy, dynamic dispatch, and speculative execution make it nearly impossible to tell

how long it will take to execute a particular piece of code. Worse, execution time is context-dependent, which leads to unmanageable variability. Still worse, programming languages usually are Turing complete, which consequently makes execution time undecidable in general.

To get predictable timing, embedded software designers must choose alternative processor architectures such as programmable DSPs, and they must use disciplined programming techniques that, for example, avoid recursion.

Engineers also stick to low-level programming because embedded software typically must interact with hardware specialized to the application. In conventional software, interaction with hardware is the operating system's domain. Typically, application designers do not create device drivers, nor do these drivers form part of an application program. In the embedded software context, however, generic hardware interfaces are rare.

Indeed, higher-level languages do not support creating interfaces to hardware. For example, although concurrency is common in modern programming languages such as Java, which has threads, no widely used programming language includes the notion of interrupts in its semantics. Yet the concept is not difficult and can be built into programming languages. For example, nesC and TinyOS, which are widely used for programming sensor networks, support interrupts at the language level.

Considering these factors, we can see that embedded software engineers do not avoid the many recent improvements in computation out of ignorance. Rather, they seek to avoid a mismatch of the core abstractions and the technologies built upon them.

In embedded software, time matters, yet computing's 20th-century abstractions hold time to be irrelevant. In embedded software, concurrency and interaction with hardware are intrinsic because embedded software engages the physical world in nontrivial ways.

The most influential 20th-century computing abstractions speak only weakly about concurrency, if at all. Even the core 20th-century notion of *computable* is at odds with the requirements of embedded software. In this notion, useful computation terminates, but termination is undecidable. In embedded software, termination is failure—yet to get predictable timing, subcomputations must decidably terminate.

In embedded software, time matters, yet computing's 20th-century abstractions hold time to be irrelevant.

TIMING'S CRUCIAL ROLE

Embedded systems consist of software and hardware integrations in which the software reacts to sensor data and issues commands to hardware actuators.

The physical system forms an integral part of the design, and the software must be conceptualized to operate in concert with it. Physical systems are intrinsically concurrent and temporal. Actions and reactions happen simultaneously and over time, and the metric properties of time play an essential part in the system's behavior.

Prevailing software methods abstract away time, replacing it with ordering. In imperative languages such as C, C++, and Java, the program defines the order of actions, but not their timing.

THE PROBLEM WITH THREADS

Another abstraction, threads or processes, overlays this prevailing imperative abstraction. The operating system typically provides this alternative abstraction, but occasionally the programming language does so.

Threads mainly focus on providing an illusion of parallelism in fundamentally sequential models, and they work well only for modest levels of concurrency or for highly decoupled systems that share resources, where best-effort scheduling policies are sufficient. Indeed, several recent innovative embedded software frameworks, such as The MathWorks' Simulink, UC Berkeley's nesC and TinyOS, and Esterel Technologies' Lustre/SCADE all provide concurrent programming languages with no threads or processes in the programmer's model.

Users generally hold embedded software systems to a much higher reliability standard than general-purpose software. Often, failures in the software can be life threatening. The prevailing concurrency model in general-purpose software does not achieve adequate reliability. This model makes it extremely difficult for humans to understand the interaction between threads. Although we can argue that concurrent computation is inherently complex, threads make it far more so because any part of the system's state can change between any two atomic operations.

The basic techniques for controlling this interaction use semaphores and mutual exclusion locks, methods that date back to the 1960s. Many uses of these techniques lead to deadlock or livelock. In general-purpose computing, this inconvenient event typically forces a program restart or even a reboot.

In embedded software, however, such errors can be far more than inconvenient. Even in general-purpose software systems, interactions with or between device drivers built on these low-level concurrency mechanisms often cause failures. Moreover, developers frequently write software without sufficiently using interlock mechanisms, which results in race conditions that yield nondeterministic program behavior.

In practice, testing cannot easily detect errors from the misuse or nonuse of semaphores and mutual exclusion locks. Code can be exercised for years before a design flaw appears.

Static analysis techniques, such as Sun Microsystems' LockLint, can help, but both conservative approximations and false positives often thwart these methods, thus they are not widely used in practice.

Reliability through clarity

We can argue that multithreaded programs' unreliability stems at least in part from inadequate software engineering processes. For example, better code reviews, specifications, compliance testing, and development process planning can help solve these problems.

Given the difficulty of understanding programs that use threads, however, no amount of process improvement will make such a program reliable if its developers cannot understand it. Formal methods can help detect flaws in threaded programs and, in the process, can improve the designer's understanding of a complex program's behavior. But if the basic mechanisms fundamentally make programs difficult to understand, these improvements will fall short of delivering reliable software.

Prevailing industrial practice in embedded software relies on bench testing for concurrency and timing properties. This has worked reasonably well because programs are small and the software is encased in a box where no outside connectivity can alter its behavior. However, applications today demand that embedded systems be feature-rich and networked, so bench testing and encasing become inadequate.

In a networked environment, it is impossible to test the software under all possible conditions because the environment is unknown. Moreover, general-purpose networking techniques themselves make program behavior much more unpredictable.

REINVENTING COMPUTER SCIENCE

Achieving concurrent and networked embedded software that can be absolutely positively on time—say, to the precision and reliability of digital logic—will, again, require sweeping changes:

- The core abstractions of computing must be modified to embrace time.
- Computer architectures must deliver precisely timed behaviors.
- The hardware–software boundary must be rethought.
- Networking techniques must provide time concurrence.
- Programming languages must embrace time and concurrency in their core semantics.
- Virtual machines must rely less on just-in-time compilation.
- Power management techniques must rely less on voltage and clock speed scaling or must couple these with timing requirements.
- Operating systems must rely less on priorities to indirectly specify timing requirements.
- Memory management techniques must account for timing constraints.
- Complexity theory must morph into schedulability analysis.
- Software engineering methods must change to specify and analyze software's temporal dynamics.
- Developers must rethink the traditional boundary between the operating system and the programming language.

Applications today demand that embedded systems be feature-rich and networked, so bench testing and encasing become inadequate.

In essence, we must reinvent computer science. Fortunately, we have quite a bit of knowledge and experience to draw upon.

Architecture techniques such as software-managed caches promise to deliver much of the benefit of memory hierarchy without the timing unpredictability. Pipeline interleaving and stream-oriented architectures offer deep pipelines with deterministic execution times. FPGAs with processor cores provide alternative hardware and software divisions.

To date, however, all these hardware techniques largely lack programming language and compiler support.

On the software side, operating systems such as TinyOS provide simple ways to create thin wrappers around hardware, and, with nesC, alter the OS/language boundary. Programming languages such as Lustre/SCADE provide understandable and analyzable concurrency. Embedded software languages such as Simulink provide time in their semantics. Bounded pause-time garbage collectors provide memory management with timing determinism.

On the networking side, time-triggered architectures provide deterministic media access and improved fault tolerance. Network time synchronization methods such as IEEE 1588 provide time concurrence at nanosecond resolutions far finer than any processor or software architectures can exploit today.

On the theory side, hybrid systems theory provides a semantics that is both physical and computational.

With so many promising starts, the time is ripe to pull these techniques together and build 21st-century embedded computer science. ■

Edward A. Lee is a professor, chair of the Electrical Engineering Division, and associate chair of Electrical Engineering and Computer Sciences at the University of California, Berkeley. Contact him at eal@eecs.berkeley.edu.

Editor: Wayne Wolf, Dept. of Electrical Engineering, Princeton University, Princeton NJ; wolf@princeton.edu