Building Unreliable Systems out of Reliable Components: The Real Time Story

Edward A. Lee

October 07, 2005

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 award #CCR-0225610), the State of California Micro Program, and the following companies: Agilent, DGIST, General Motors, Hewlett Packard, Infineon, Microsoft, and Toyota.

Building Unreliable Systems out of Reliable Components: The Real Time Story

Edward A. Lee
eal@eecs.berkeley.edu

Professor, Chair of the Electrical Engineering Division, and
Associate Chair of Electrical Engineering and Computer Sciences
University of California, Berkeley

Abstract of Invited Plenary Talk
Monterey Workshop
Laguna Beach, California
September 22, 2005

Despite considerable progress in software and hardware techniques, when embedded computing systems absolutely must meet tight timing constraints, many of the advances in computing become part of the problem rather than part of the solution. The underlying technology for computation, synchronous digital logic, easily delivers precise timing determinacy (although certain deep submicron techniques threaten even this foundation). However, advances in computer architecture and software have made it difficult or impossible to estimate or predict the execution time of software. Moreover, networking techniques introduce variability and stochastic behavior, and operating systems rely on best effort techniques. Worse, programming languages lack time in their semantics, so timing requirements are only specified indirectly. I examine the following question: "if precise timeliness in a networked embedded system is absolutely essential, what has to change?" The answer, unfortunately, is "nearly everything."

Twentieth century computer science has taught us that everything that can be computed can be specified by a Turing machine. "Computation" is accomplished by a terminating sequence of state transformations. A "Computable Function" is a map from a bit sequence to a bit sequence. This core abstraction underlies the design of nearly all computers, programming languages, and operating systems in use today. But unfortunately, this core abstraction does not fit embedded software very well, and since it says nothing about timeliness, technologies for computation need not be concerned with timeliness.

This core abstraction fits reasonably well if embedded software is simply "software on small computers." 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 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.

Of course, thanks to the semiconductor industry's ability to follow Moore's law, the resource limitations of 25 years ago should have almost entirely evaporated today. Why then has embedded software design and development changed so little? It may be that extreme competitive pressure in products based on embedded software, such as consumer electronics, rewards only the most efficient solutions. This argument is questionable, however. There are many examples where functionality and reliability have proven more important than efficiency. It is arguable that resource limitations are not the only defining factor for embedded software, and may not even be the principal factor.

There are clues that embedded software differs from other software in more fundamental ways. If we examine carefully why engineers write embedded software in assembly code or C, we discover that efficiency is not the only concern, and may not even be the main concern. The reasons may include, for example, 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 are about scalability (inheritance, dynamic binding, polymorphism, memory management), and, if anything, further obscure timing (consider the impact of garbage collection on timing). Counting cycles, of course, becomes extremely difficult on modern processor architectures, where memory hierarchy (caches), 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 are almost always Turing complete, and as a consequence, execution time is undecidable in general. Embedded software designers must choose alternative processor architectures such as programmable DSPs, and must use disciplined programming techniques (e.g. avoiding recursion) to get predictable timing.

Another reason engineers stick to low-level programming is that embedded software typically has to interact with hardware that is specialized to the application. In conventional software, interaction with hardware is the domain of the operating system. Device drivers are not typically part of an application program, and are not typically created by application designers. But in the embedded software context, generic hardware interfaces are rarer. The fact is that creating interfaces to hardware is not something that higher level languages support. For example, although concurrency is not uncommon in modern programming languages (consider threads in Java), no widely used programming language includes in its semantics the notion of interrupts. Yet the concept is not difficult, and it can be built into programming languages (consider for example nesC and TinyOS, which are widely used for programming sensor networks).

It becomes apparent that the avoidance by embedded software engineers of so many recent improvements in computation is not due to ignorance of those improvements. It is due to a mismatch of the core abstractions and the technologies built on those core abstractions. In embedded software, time matters. In the 20th century abstractions of computing, time is irrelevant. In embedded software, concurrency and interaction with hardware are intrinsic, since embedded software engages the physical world in non-trivial ways (more than keyboards and screens). 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, and yet to get predictable timing, subcomputations must decidably terminate.

Embedded systems are integrations of software and hardware where the software reacts to sensor data and/or issues commands to actuators. The physical system is an integral part of the design and the software must be conceptualized to operate in concert with that physical system. Physical systems are intrinsically concurrent and temporal. Actions and reactions happen simultaneously and over time, and the metric properties of time are an essential part of the behavior of the system. Prevailing software methods abstract away time, replacing it with ordering. In imperative languages such as C, C++, and Java, the order of actions is defined by the program, but not their timing. This prevailing imperative abstraction is overlaid with another, that of threads or processes, typically provided by the operating system, but occasionally by the language (as in Java).

The lack of timing in the core abstraction is a flaw, from the perspective of embedded software, and threads as a concurrency model are a poor match for embedded systems. They are mainly focused 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 are sharing resources, where best-effort scheduling policies are sufficient. Indeed, several recent innovative embedded software frameworks, such as Simulink (from The MathWorks), nesC and TinyOS (from Berkeley), and Lustre/SCADE (from Esterel Technologies) are concurrent programming languages with no threads or processes in the programmer's model.

Embedded software systems are generally held to a much higher reliability standard than general purpose software. Often, failures in the software can be life threatening (e.g., in avionics and military systems). The prevailing concurrency model in general purpose software that is based on threads does not achieve adequate reliability. In this prevailing model, interaction between threads is extremely difficult for humans to understand. Although it is arguable that concurrent computation is inherently complex, threads make it far more complex because between any two atomic operations (a concept that is rarely well defined), any part of the state of the system can change. 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 is inconvenient, and typically forces a restart of the program (or even a reboot of the machine). However, in embedded software, such errors can be far more than inconvenient. Even in general-purpose software systems, failures are often caused by interactions with or between device drivers, which are built on these low-level concurrency mechanisms. Moreover, software is often written without sufficient use of these interlock mechanisms, resulting in race conditions that yield nondeterministic program behavior. In practice, errors due to misuse (or no use) of semaphores and mutual exclusion locks are extremely difficult to detect by testing. Code can be exercised for years before a design flaw appears. Static analysis techniques can help (e.g. Sun Microsystems' LockLint), but these methods are often thwarted by conservative approximations and/or false positives, and they are not widely used in practice.

It can be argued that the unreliability of multi-threaded programs is due at least in part to inadequate software engineering processes. For example, better code reviews, better specifications, better compliance testing, and better planning of the development process can help solve the problems. It is certainly true that these techniques can help. However, programs that use threads can be extremely difficult for programmers to understand. If a program is incomprehensible, then no amount of process improvement will make it reliable. Formal methods can help detect flaws in threaded programs, and in the process can improve the understanding that a designer has of the behavior of a complex program. But if the basic mechanisms fundamentally lead to programs that are difficult to understand, then these improvements will fall short of delivering reliable software. Incomprehensible software will always be unreliable 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 because the software gets encased in a box with no outside connectivity that can alter the behavior of the software. However, applications today 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, because the environment is not known. Moreover, general-purpose networking techniques themselves make program behavior much more unpredictable.

What would it take to achieve concurrent and networked embedded software that was **absolutely positively on time** (say, to the precision and reliability of digital logic)? Unfortunately, everything would have to change. The core abstractions of computing need to be modified to embrace time. Computer architectures need to be changed to deliver precisely timed behaviors. The hardware/software boundary needs to be rethought. Networking techniques need to be changed to provide time concurrence. Programming languages have to change to embrace time and concurrency in their core semantics. Virtual machines have to change to rely less on just-in-time compilation. Power management techniques need to change to rely less on voltage and clock speed scaling, or to couple these with timing requirements. Operating systems have to change to rely less on priorities to (indirectly) specify timing requirements. Memory management techniques need to account for timing constraints. Complexity theory needs to morph into schedulability analysis. Software engineering methods need to change to specify and analyze the temporal dynamics of software. And the traditional boundary between the operating system and the programming language needs to be rethought. What is needed is nearly a reinvention of computer science.

Fortunately, there is quite a bit to draw on. To name a few examples, architecture techniques such as software-managed caches (scratchpad memories) 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/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 medium access and improved fault tolerance. Network time synchronization methods such as IEEE 1588 provide time concurrence at resolutions (nanoseconds) 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 the 21st Century (Embedded) Computer Science.