What are the Key Challenges in Embedded Software?

Edward A. Lee

Guest Editorial
in System Design Frontier, Volume 2, Number 1, January 2005
Shanghai Hometown Microsystems Inc http://www.hwswworld.com/

Embedded software has traditionally been thought of as "software on small computers." In this traditional view, the principal problem is resource limitations (small memory, small data word sizes, and relatively slow clocks). Solutions emphasize efficiency; software is written at a very low level (in assembly code or C), operating systems with a rich suite of services are avoided, and specialized computer architectures such as programmable DSPs and network processors are developed to 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, since there are many examples where functionality has proven more important than efficiency. We will argue that resource limitations are not the only defining factor for embedded software, and may not even be the principal factor.

Resource limitations are an issue to some degree with almost all software. So generic improvements in software engineering should, in theory, also help with embedded software. There are several hints, however, that embedded software is different in fundamental ways. For one, object-oriented techniques such as inheritance, dynamic binding, and polymorphism are rarely used in practice with embedded software development. In another example, processors used for embedded systems often avoid the memory hierarchy techniques that are used in general purpose processors to deliver large virtual memory spaces and faster execution using caches. In a third example, automated memory management, with allocation, deallocation, and garbage collection, are largely avoided in embedded software. To be fair, there are some successful applications of these technologies in embedded software, such as the use of Java in cell phones, but their application remains limited and is largely providing services that are actually more akin to general-purpose software applications (such as database services in cell phones).

Embedded systems are integrations of software and hardware where the software reacts to sensor data and 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 to embedded systems. They are mainly focused on providing an illusion of concurrency 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), TinyOS (from Berkeley), and SCADE (from Esterel Technologies) have no threads or processes.

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 based on threads does not achieve adequate reliability. In this prevailing model, interaction between threads is extremely difficult for humans to understand. The basic techniques for controlling this interaction use semaphores and mutual exclusion locks, methods that date back to the 1960s. These techniques often lead to deadlock or livelock conditions, where all or part of a program cannot continue executing. 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. 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.

The key challenge in embedded software is to invent (or apply) abstractions that yield more understandable programs that are both concurrent and timed. These abstractions will be very different from those widely used for the design and development of general-purpose software.