

**EMBEDDED-SYSTEM PROGRAMMERS CAN NO LONGER RELY ON MOORE'S LAW. ITS DIMINISHING EFFECT ON PROCESSOR PERFORMANCE DOESN'T CONCEAL INEFFICIENT AND DISORGANIZED CODE.**

# Embedded-system programmers must learn the fundamentals

**C**PU POWER HAS FOR THE PAST three decades dictated the way software engineers design and implement embedded-software options. During the early days of the microprocessor revolution, programmers had to use every trick in the book to shoehorn simple applications onto low-frequency, 4- and 8-bit microcontrollers. While their data-processing peers were busy enjoying the software-engineering benefits of high-level languages and capacious systems, embedded-system designers were still worrying about making every byte and CPU cycle count. Computer-application-level software engineers often get to work with the latest and greatest hardware, but embedded-system engineers usually have to make do with much lower performing hardware. It's unsurprising, therefore, that embedded-system engineers are among the best at doing the most work with a given amount of processing power.

Embedded-software-development practices seem to lag by about a decade behind commercial-software-development practices. When commercial-system programmers began in the mid-1980s switching to high-level languages, embedded-system programmers continued to work in assembly language. When commercial-system programmers switched to higher level languages, such as C++ and Java, embedded-system programmers began using C. Though improvements in compiler technology have helped drive this situation, the move from low- to high-level programming is largely due to the performance and economic side effects of Moore's Law and the passage of four decades since its discovery.

Until recently, the memory capacity of modern computer systems has doubled every couple of years, the performance has also nearly doubled, and the cost of such systems has come down. Programmers know these effects and side effects of Moore's Law so well that most of them not only recognize, but also depend on this progression for successful deployment of their software systems. That is, they count on the fact that computer systems will run faster and offer greater capacity by the time the programmers complete the first version of their software to fulfill performance requirements. However, assuming that

CPU performance and capacity will always improve at an exponential rate presents problems.

The assumption that CPU performance will soon double has allowed programmers to avoid optimizing their code. As a result, many programmers have never learned how to write optimal code, and others have allowed their skills to atrophy.

Also, Moore's Law does not predict that performance will double every couple of years; it predicts only that the number of transistors in an IC will double every two years. Until recently, CPU designers have been able to use those extra transistors to dramatically improve performance. Lately, however, raw CPU performance has not been increasing at the same rate as it has in the past. Today, you see 15% jumps in clock frequencies rather than the 200% increases of 10 years ago in each new CPU generation (**Reference 1**).

Finally, exponential improvement cannot continue, and Moore's Law is running out of steam. Gordon Moore himself predicts about another decade before physical reality sets in.

## NO SILVER BULLET

Fundamentally, the end of Moore's Law will have a tremendous impact on the software-development community. Software engineers will no longer be able to assume that performance and capacity problems are the domain of semiconductor designers. Instead of relying on the hardware designers to provide a silver bullet, programmer's must now deliver applications that meet performance specifications.

The only wrinkle here is that a new generation of software engineers has grown up learning their craft on high-performance computer systems, writing code in abstract languages such as Java, PERL (practical extraction and reporting language), and Python. Experts have warned these programmers, throughout their educational and professional and professional lives, to avoid machine dependency at all costs. As a result, many of today's graduating students and practicing software engineers lack a fundamental knowledge of low-level-computer-system operation, which is crucial if they intend to write op-

timized code for their applications.

British computing pioneer Sir Tony Hoare once wrote: "Premature optimization is the root of all evil." Unfortunately, engineers often take this phrase out of context and use it to justify avoiding any thought of optimization or even plans to optimize in their code.

Charles Cook succinctly explains the problem with this approach: "The full version of the quote is 'We should forget about small efficiencies—say, about 97% of the time: Premature optimization is the root of all evil,' and I agree. It's usually not worth spending a lot of time micro-optimizing code before it's obvious where the performance bottlenecks are. But, conversely, when designing software at a system level, performance issues should always be considered from the beginning. A good software developer will do this automatically, having developed a feel for where performance issues will cause problems. An inexperienced developer will not bother, misguidedly believing that a bit of fine-tuning at a later stage will fix any problems."

The key point here is that inexperienced developers often write code without any consideration of the performance of their code. Unfortunately, system design without any concerns about performance rarely produces systems that perform well without major rewriting. The only thing worse than premature optimization is designing a system without any consideration of system performance. The assumption that 20% of a program's code accounts for 80% of its execution time has been the downfall of many designs.

"Programmer productivity" has been the mantra ever since in the late 1960s someone coined the term "software crisis." Software engineers have in the past got away with taking the easy way out because ever-increasing CPU performance has covered up for poor engineering. This scenario is no longer the case. Software engineers no longer have the luxury of writing less than great code and having ever-increasing CPU performance. Software engineers must supply applications that meet performance specifications.

### **STOP THE WASTE**

The good news, however, is that programmers have in the past been wasteful of CPU performance. With education

## **THE INDUSTRY WINDS UP WITH EMBEDDED-SYSTEM PROGRAMMERS WHO DON'T UNDERSTAND THE DIFFERENCES BETWEEN FLOATING- AND FIXED-POINT ARITHMETIC OR HOW TO EFFICIENTLY PASS PARAMETERS TO A FUNCTION.**

and better engineering, programmers can easily improve the performance of their software systems without overtly affecting their productivity. They may require some education, and that education consumes time that they could possibly use on some project. However, continuing education is a major expectation for engineers, so learning how to write better code is something that engineers should consider simply part of their professional lives.

So, what do engineers need to learn to write great code? The first place to start is with the fundamentals. One piece of fundamental knowledge is machine organization—that is, understanding how the machine operates. Any embedded-system engineer should have this knowledge. Recent job postings on Internet job-posting sites, such as Monster.com, make statements such as, "must be able to work with low-level hardware and know assembly language." This scenario suggests that far too many Java programmers are calling themselves "embedded-system engineers" and feel that the purpose of high-level languages is to shield them from ever having to master details, such as machine organization.

Old-time embedded-system engineers may find this notion—that anyone would call himself an embedded-system engineer without understanding the hardware—absurd, but it is becoming more commonplace with each graduating class. The problem is that engineering schools typically teach machine organization as part of a curriculum's assembly-language-programming course, and the instructor often begins the course by saying: "This material is obsolete, and you'll never use it, but you need it to graduate, so here we go." Is it no wonder that students quickly forget this material? The result is that the industry winds up with embedded-system programmers who

don't understand the differences between floating- and fixed-point arithmetic or how to efficiently pass parameters to a function. Students walk away from computer-science programs believing two code implementations have the same performance because only a multiplicative constant difference in performance exists between the two. In other words, code running just two times faster for all inputs is insignificant. Such students are doomed to writing low-quality code until they master this fundamental knowledge.

Although hardware designers may not be able to provide us with faster and faster CPUs, there is some good news. Because programmers have not recently been writing optimized code, ample opportunity exists for improving embedded-application performance by using better programming techniques and methodologies. Programmers can implement these methods without dropping down into assembly language or returning to difficult optimization techniques. The first step is to begin thinking in low-level terms but writing high-level code. By simply understanding how typical CPUs execute machine-level code and how compilers translate high-level source statements into machine instructions, programmers can wisely choose which high-level-language statements produce the best possible machine-instruction sequence. Every programmer should know these fundamentals, but schools and projects haven't emphasized them because the hardware has always come to the rescue.

### **MASTERING MACHINE ORGANIZATION**

The fact that many inexperienced programmers have failed to master fundamental knowledge, such as machine organization, is a problem. To solve the problem, try to ensure the mastery of machine organization in new employees. One way of accomplishing this task is to suggest that knowing assembly language is a plus for the position. Even if you would never allow the use of assembly language in your project, if an applicant knows assembly language, then he probably has a grasp of machine organization. Suggesting that an applicant needs to know assembly language helps encourage that student to pay more attention in their machine-organization and assembly-language courses.

Also, senior engineers should spend time mentoring junior engineers. The craftsman/apprentice model is particularly applicable here. All too often, employers assign fresh graduates to a project without having senior engineers properly guide them. Although they may be eager and energetic, the students' inexperience often leads them to mediocre code. Senior engineering staff should carefully review their work and explain why certain options are better than others for the project. By reviewing the students' code and showing them how to improve it, senior engineers can help inexperienced programmers become great programmers.

Also, counter the myth that performance is irrelevant and that optimization is a waste of time. Though premature optimization can be bad, designing code without any thought to its ultimate optimization often yields code that is difficult, if not impossible, to optimize. Too many programmers feel that performance is a hardware issue. Unfortunately, programmers holding such views are of-

ten the primary causes of the problem. Companies need to eliminate the culture that believes performance is someone else's problem.

Software engineers who are uncomfortable with fundamental topics, such as machine organization, should rectify this situation. Learning an assembly language is a way to master machine organization; learning assembly is sometimes not a viable option. In that case, engineers should consider reading a book, of which there are several, that directly teaches machine organization without the overhead of assembly.

Recently, software engineers have relied on ever-faster hardware to solve their performance problems. The free ride, however, is quickly coming to an end. The hardware designers have given their all; now, it's the programmer's responsibility to provide high-performance options. You can write efficient code that you can maintain and deliver on time. But you have to plan on this scenario from the beginning. As with any engineering endeavor, if you properly plan

things from the start, you need not deal with surprises later. Those who write code with no concern for efficiency should be unsurprised when their code is sluggish. A little foresight, planning, and education are all it takes to avoid this problem. □

---

#### REFERENCES

1. [www.gotw.ca/publications/concurrency-ddj.htm](http://www.gotw.ca/publications/concurrency-ddj.htm).
2. [www.cookcomputing.com/blog/archives/000084/html](http://www.cookcomputing.com/blog/archives/000084/html).

---

#### AUTHOR'S BIOGRAPHY

*Randall Hyde is the author of Write Great Code: Understanding the Machine (No Starch Press) and co-author of the Waite Groupe's MASM Bible. He has written for Dr. Dobb's Journal, Byte, and other professional journals. He also works as an embedded-system-software engineer, writes device drivers, and develops hardware.*

---

#### TALK TO US

*Post comments via TalkBack at the online version of this article at [www.edn.com](http://www.edn.com).*

---