Moore's Law makes it seem as if resource limitations are always a minor consideration. If there will be twice as much memory for the same price in 18 months, why bother to squeeze a factor of 2 from an application's memory requirements? If the CPU will be twice as fast by then, why bother to shave some running time from a program? In other words, why bother to optimize programs? Isn't it better to just get them running and let Moore's Law take us off the hook when resources are constrained?

Randall Hyde argues that optimization is important even when memory and processor double regularly. Trying to do the optimization too early can be a futile time-waster. His provocative analysis deals with an issue that just won't go away.

Peter Denning
Editor

The Fallacy of Premature Optimization

by Randall Hyde

Every programmer with a few years' experience or education has heard the phrase "premature optimization is the root of all evil." This famous quote by Sir Tony Hoare (popularized by Donald Knuth) has become a best practice among software engineers. Unfortunately, as with many ideas that grow to legendary status, the original meaning of this statement has been all but lost and today's software engineers apply this saying differently from its original intent. As computer systems increased in performance from MHz, to hundreds of MHz, to GHz, the performance of computer software has taken a back seat to other concerns. Today, it is not at all uncommon for software engineers to extend this maxim to "you should never optimize your code!" Funny, you don't hear too many computer application users making such statements. It is unfortunate that Hoare's comments have been twisted to imply that optimization is unnecessary. The bloat and unresponsiveness found in many modern applications compels software engineers to reconsider how they apply Hoare's comments to their projects.

As you can probably tell, this article is not "yet another article warning beginning programmers to avoid premature optimization." The purpose of this article is to examine how software engineers have (incorrectly) applied Hoare's statement as a way of avoiding the effort necessary to produce a well-performing application. Hopefully, this article can encourage many software engineers to change their views on application performance.

"Premature optimization is the root of all evil" has long been the rallying cry by software engineers to avoid any thought of application performance until the very end of the software development cycle (at which point the optimization phase is typically ignored for economic/time-to-market reasons). However, Hoare was not saying, "concern about application performance during the early stages of an application's development is evil." He specifically said premature optimization; and optimization meant something considerably different back in the days when he made that statement. Back then, "optimization" often consisted of activities such as counting cycles and instructions in assembly language code. This is not the type of coding you want to do during initial program design, when the code base is rather fluid. So Hoare's comments were on the mark. Indeed, a short essay by Charles Cook (http://www.cookcomputing.com/blog/archives/000084.html), part of which I've reproduced below, describes the problem with reading too much into Hoare's statement:

I've always thought this quote has all too often led software designers into serious mistakes because it has been applied to a different problem domain to what was intended. 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 with this. Its usually not worth spending a lot of time micro-optimizing code before its 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.

What Hoare and Knuth are really saying is that software engineers should worry about other issues (such as good algorithm design and good implementations of those algorithms) before they worry about micro-optimizations such as how many CPU cycles a particular statement consumes.

It is interesting to look at how some software engineers have perverted those good software engineering concepts to avoid the work associated with writing efficient code.

Observation #1: "Premature optimization is the root of all evil" has become "Optimization is the root of all evil." Therefore, optimization should be avoided.

Observation #2: Many software engineers believe that optimization is the act of ensuring an application has adequate performance. As a result, those engineers do not consider application performance during the design of the software, when it is critical to do so.

Observation #3: Software engineers use the Pareto Principle (also known as the "80/20 rule") to delay concern about software performance, mistakenly believing that performance problems will be easy to solve at the end of the software development cycle. This belief ignores the fact that the 20 percent of the code that takes 80 percent of the execution time is probably spread throughout the source code and is not easy to surgically modify. Further, the Pareto Principle doesn't apply that well if the code is not well-written to begin with (i.e., a few bad algorithms, or implementations of those algorithms, in a few locations can completely skew the performance of the system).

Observation #4: Many software engineers have come to believe that by the time their application ships CPU performance will have increased to cover any coding sloppiness on their part. While this was true during the 1990s, the phenomenal increases in CPU performance seen during that decade have not been matched during the current decade.

Observation #5: Software engineers have been led to believe that they are incapable of predicting where their applications spend most of their execution time. Therefore, they don't bother improving performance of sections of code that are obviously bad because they have no proof that the bad section of code will hurt overall program performance.

Observation #6: Software engineers have been led to believe that their time is more valuable than CPU time; therefore, wasting CPU cycles in order to reduce development time is always a win. They've forgotten, however, that the application users' time is more valuable than their time.

Observation #7: Optimization is a difficult and expensive process. Many engineers argue that this process delays entry into the marketplace and reduces profit. This may be true, but it ignores the cost associated with poor-performing products (particularly when there is competition in the marketplace).

Observation #8: The most fundamental rule software engineers cite when performance is a concern is "choosing the proper algorithm is far more important than any amount of (micro-) optimization you can do to your code." This ignores the fact that a better algorithm might not be available or may be difficult to discover or implement. In any case, this is not a good excuse for creating a poor implementation of any algorithm.

Observation #9: There is little need to ensure that you have the best possible algorithm during initial software design because you can always substitute a better algorithm later. Therefore, there is no need to worry about performance during initial software design because it can always be corrected with a better algorithm later. Unfortunately, people who take this approach to software design often write code that cannot be easily modified down the road.

So how do we reverse the trend and convince software engineers that they should pay more than lip service to writing efficient applications? Largely, what is necessary is a change in mindset; programmers must be convinced that efficiency is an important criterion when developing software.

The first place to start is with Hoare's original saying. He did not say, "Optimization is the root of all evil." So software engineers must accept optimization as a necessary step and not reject it as evil. Optimization is a standard part of the software development process, along with testing, debugging, and quality assurance, and it should be treated as such. Yes, optimization is difficult and requires considerable experience to do it properly. However, avoidance is not the solution to a difficult problem. The only way to gain the experience needed to properly optimize code is by constant practice. Testing is difficult and time-consuming too; you would never see

a professional software engineer seriously suggesting that testing isn't necessary because it is difficult or time-consuming.

The second thing to note is that Hoare did not simply say, "Premature optimization is the root of all evil." He said, "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." Small efficiencies (that is, the kind of stuff you get with micro-optimization) typically aren't worth the effort during the initial design and implementation of a system. Note, however, that Hoare did not say, "Forget about small efficiencies all of the time." Instead, he said "about 97% of the time." This means that about 3% of the time we really should worry about small efficiencies. That may not sound like much, but consider that this is 1 line of source code out of every 33. How many programmers worry about the small efficiencies even this often? Premature optimization is always bad, but the truth is that some concern about small efficiencies during program development is not premature. How does one determine which optimizations are worthy of consideration during development and which would be premature? Charles Cook's quote was right on the mark: "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." So how does an inexperienced programmer learn when early optimization is appropriate? The first step is to profile everything; only by timing sections of the program can a programmer learn to predict the performance of similar code segments. The second thing an inexperienced programmer must do is be willing to make some mistakes. One thing nice about optimization is that if you optimize a section of code that doesn't need it, you've not done much damage to the application. Other than possible maintenance issues, all you've really lost is some time optimizing code that doesn't need it. Though it might seem that you've lost some valuable time unnecessarily optimizing code, don't forget that you have gained valuable experience so your are less likely to make that same mistake in a future project.

The third necessary adjustment in practice that software engineers should make is to pay more than lip service to designing performance into an application from the very start. The whole concept of "premature optimization is the root of all evil" is contrary to the software engineering maxim design first, code second. Systems that are designed without any thought of performance rarely perform well, even with an extensive optimization phase at the end of the software development cycle. Often, the only way to improve such systems is by rewriting them. Too many engineers use Hoare's saying as an excuse to avoid the difficult work of designing and implementing high-performance code. However, Hoare did not say, "Efficiency is the root of all evil." The design and optimization phases are completely separate and Hoare's saying applies only to the optimization phase, not the design phase. If premature optimization is the root of all evil, then the lack of planned performance during the design and implementation phases is the trunk, branches, and leaves of all evil. Premature optimization may lead to code that is difficult to modify, maintain, and read. Improper design and implementation, on the other hand, often necessitate a complete rewrite. Programmers must also master their tools and understand their behavior. A favorite saying of mine, which I learned from Rico Mariani (http://msdn.microsoft.com/en-us/library/ms979190.aspx) is "Never give up your performance accidentally." It is amazing how often the typical software engineer violates this when writing source code. For example, many programmers don't have a clue concerning the cost of the high-level language statements they use in their source code. Most of them assume, as they were taught in their college Data Structures and Algorithm Analysis course, that each statement in a program takes one unit of time to execute. In reality, different statements require differing amounts of execution time and sometimes the choice of one type of statement over another can make a difference in the execution time of the program. Now it is never a good idea to choose the control statements in a program based solely on their performance, but given two otherwise equivalent statements, choosing the more efficient one is usually the wise choice. Unfortunately, few programmers really understand the implementation details, so they aren't able to choose which statement(s) are most appropriate for their code. This is a classic example of accidentally giving up performance.

Traditionally, software engineers have used the excuse "premature optimization is the root of all evil" to justify their lack of attention to implementation details such as choosing the most efficient of two (or more) equivalent instruction sequences. Their argument typically takes the form of "it takes too much time to write code that way, especially when that code is likely to change within a few days." However, experienced programmers do not spend much extra time deciding which of several possible instruction sequences to use; by experience, they know which sequence is best and tend to go with that choice, giving it very little extra thought. This should be the goal of every software engineer-to get to the point where they automatically choose a good (if not the best) implementation for a given code sequence. Alas, all too often, a typical software engineer will choose whatever sequence first comes to mind, or whatever sequence seems most expedient at the time. Unfortunately, if the engineer doesn't consider the quality of the code they're writing, the resulting random choice is more likely to be bad rather than good, simply because there are so many more bad choices than good ones.

To ensure high-performance in a high-quality application isn't as difficult as many software engineers would have you believe. It doesn't cost a lot of extra money to develop high-performance software nor does the development of high-performance software inordinately delay the release of that software. To achieve this, system designers must design performance into the product from the beginning; programmers need to carefully consider how they implement the designs when coding the application; and the quality assurance team needs to check the performance of the product during development to assure that its performance is acceptable. It is less costly to correct performance issues, just like any other defect, as early in the development cycle as possible. That's why putting off concern about efficiency until the optimization phase is usually a mistake.

So what should a software engineer study if they want to learn how to write efficient code? Well, the first subject to master is machine organization (elementary computer architecture). Because all real programs execute on real machines, you need to understand how real machines operate if you want to write efficient code for those machines. Machine organization covers subjects such as data representation, memory access and organization, cache operation, composite data structure implementation, how computers perform arithmetic and logical operations, and how machines execute instructions. This is fundamental knowledge that impacts almost every aspect of the implementation of a software system.

The second subject to study is assembly language programming. Though few programmers use assembly language for application development, assembly language knowledge is critical if you want to make the connection between a high-level language and the low-level CPU. I'm not going to suggest that all programmers should become expert assembly language programmers in order to write great code, but high-level language programmers who have a firm understanding of the underlying machine code have written some of the best code in existence.

Note that for the purposes of understanding the costs associated with the execution of high-level language statements, it doesn't really matter which assembly language you learn nor does it matter which CPU's instruction set you study. What you really need to learn are the basic operational costs of computation. Whether you're moving data around on a PowerPC or an Intel x86, the costs are roughly comparable from a high-level language perspective. Using instruction timing knowledge at a finer level of granularity than this definitely falls into the "premature optimization" area that Hoare has warned everyone about.

The main reason programmers should learn assembly language is to be able to understand the costs associated with the statements they use in day-to-day programming. Learning assembly language is an important first step, but it is not sufficient. The third important subject a software engineer should study is basic compiler construction, to learn how compilers translate high-level language statements into machine code

It is important to remember what Rico Mariani said, "Never give up your performance accidentally." Notice he did not say, "Never give up performance." He said programmers should not give up performance accidentally. People accidentally give up performance when they don't understand the costs associated with the constructs they are using in their programs. By studying how systems translate or interpret high-level language statements, you understand the costs and will not use inefficient constructs because you don't know any better. Instead, you can compare the cost of using a given construct against other methods that can achieve the same result and choose the most appropriate implementation on the basis of many factors, including readability, maintainability, and performance. The choice might not always be the most efficient implementation. However, if you choose to give up performance (to improve readability or maintainability, for example), you will be making an informed decision, the loss isn't accidental.

Sir Tony Hoare's statement "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil" has been responsible for a fundamental change in the way software engineers develop applications. A misinterpretation of this statement, particularly the clause "premature optimization is the root of all evil" has led many software engineers (and schools teaching new engineers) to believe that optimization and, indeed, concern about software efficiency, is unimportant. To reverse the decline in application performance, software engineers must first reject the prevailing attitude that performance is not important. Once they decide that writing efficient software is worthwhile, the next step is to learn how high-level language compilers and interpreters process those high-level constructs; once the engineer understands this process, choosing appropriate high-level constructs will become second nature and won't incur additional development cost. Hoare's statement, in its original context, certainly still applies. Premature optimization, at the microscopic level, is usually a bad idea. This does not suggest, however, that engineers should not be

concerned about application performance. The fallacy that "premature optimization" is the same thing as "concern about performance" should not guide software development.

Want to learn more? Here are a few suggestions for books covering topics that practicing software engineers should master:

Machine Organization The topic of machine organization is how machines (computers) operate and how they represent data objects. There are several good texts you might want to consider to expand your knowledge of this subject:

Computer Organization and Design: The Hardware/Software Interface. Hennessy and Patterson (Morgan Kaufmann, Aug 2004). Mixed on-line reviews, but Hennessy and Patterson are very well regarded in the area of Computer Architecture and this book is certainly worthy of consideration.

Code. Charles Petzold (Microsoft Press, Oct 2000) A low-level look at how computers operate and represent data.

Write Great Code, Volume 1: Understanding the Machine. Randall Hyde (No Starch Press, 2004) This is a book from my Write Great Code series that concentrates on machine organization.

Assembly Language. Though assembly language's heyday has passed, there are several good assembly language books still in print today. Here are a few I can recommend:

Assembly Language Step-By-Step: Programming with DOS and Linux. Jeff Duntemann (Wiley, 2000)

The Art of Assembly Language. Randall Hyde (No Starch Press, 2003).

Of course, Intel's and AMD's on-line processor reference manuals are also important for those working with x86 assembly language (or the respective manufacturer's manuals for other CPUs).

Compiler Design. For years, the standard textbook on compiler construction has been Compilers: Principles, Techniques, and Tools. Aho, Sethi, and Ulman (Addison Wesley, 1986); despite its age, this is still an excellent choice for learning basic compiler technology even if it doesn't cover the latest and greatest optimizations.

Implementation of High-Level Control Constructs. If learning compiler construction seems a little too heavy-duty, an alternative is to pick up a book on reverse engineering or disassembly; most such texts spend a fair amount of time discussing the translation of high-level language code into machine code (so the reader can reverse this process). Here are a couple of books that treat this subject:

Write Great Code, Volume 2: Thinking Low-Level, Writing High Level Randall Hyde (No Starch Press, 2006)

Reversing: Secrets of Reverse Engineering. Eldad Eilam (Wiley, 2005) Hacker Disassembling Uncovered. Kris Kaspersky (A-List Publishing, 2003)

Randall Hyde us the author of many books, including "Write Great Code, Volume 2: Thinking Low-Level, Writing High-Level," recently published by No Starch Press.