Notes on Code Optimization

Robert J. Hansen

2 October 2013

This document is copyright (c) 2013, Robert J. Hansen, and is made available under the Creative Commons Attribution Sharealike License (version 3.0, United States jurisdiction).

Abstract
Recently I've encountered several people contorting their code in ever more improbable ways in order to improve its performance. They followed no real methodology, just guesswork and "well it's gotta be here". This brief technical report will hopefully serve to help those who find themselves in a similar situation.

A Process for Optimization

Introduction

ACCORDING TO DONALD KNUTH, the vast majority of a non-I/O bound program's execution time will occupy about three percent of its overall source code. Further, it is extraordinarily difficult to make accurate predictions about where one will find this three percent.[1]

[1] Knuth, Donald E. *Structured Programming with Goto Statements.* In *Computing Surveys* 6:4 (December 1974), pages 261-301.

The upshot of this is twofold.

First, without an empirical understanding of where the bottleneck is in a piece of code it is exceedingly unlikely that changes to a codebase will have a positive effect.

Second, and just as importantly, code performance simply does not matter for the great majority of the codebase. In these non-critical sections code should be evaluated according to criteria other than performance (clarity, reliability, simplicity, etc.).

The remainder of this note will all be focused around one of these two points.

Preliminaries

WHENEVER POSSIBLE, optimization should begin from a codebase that functions correctly. Although few (if any!) programs can truly be said to be bug-free,[2] one should avoid optimizing code that is known to contain serious errors. A program's correctness is the ultimate test of its performance: a correct answer reached tomorrow is better than a wildly inaccurate answer reached in a few milliseconds.

[2]

"Beware of bugs in the above code; I have only proved it correct, not tried it."

Donald Knuth

A FEW WORDS ABOUT WRITING GOOD CODE is therefore in order. Correctness is more important than speed and clarity is more important than cleverness. Typically, fixing a bug demands a deeper understanding than writing the bug. For this reason it is good to code simply and clearly. Cleverness should shine through during bug-hunting, not during coding.

However, we live in a real world with real constraints of time, money and resources. When given an extant codebase of snarled spaghetti, it is improper to start the code optimization process by saying, "this code must be simplified and clarified." The bottleneck must be simplified and clarified, yes, but not the whole of the codebase. It is easy to defend one's decision to simplify and clarify the bottleneck, but to simplify and clarify the other 97% of the codebase may be a waste of limited resources. If management wanted the entire codebase overhauled they would have ordered that; instead, they ordered optimizations.

A HEALTHY UNDERSTANDING of what profiling tools are available for the platform in question, as well as how to use them and what reference manuals may exist for them, is essential.

Collecting data

THE MOST IMPORTANT RULE of collecting data about runs of a program is to collect data on *realistic* runs. Programmers are fond of using cooked data sets which explore rare edge cases and cause different kinds of defects to come to light. These must be avoided. These are excellent tools for software quality analysis but terrible ones for discovering where customers are likely to encounter slowdowns. If at all possible, use data sets provided by customers — and not just one customer, either. Different customers are likely to have different inputs that will tickle different parts of the program. If data sets from multiple customers all reveal a bottleneck in the same place, it's an excellent bet the bottleneck is in that location. If only one customer finds a bottleneck in a certain location, though, there may be some room for further investigation: why is this bottleneck there? What is this customer's data set tickling?

Another important rule for collecting data is to collect only the minimum data needed at a given time. Many profiling tools will spam the user with an inappropriate level of detail, making it hard to see the forest for all the trees. Dial the output level back as far as possible to get a high-level view of the problem, and then zoom in slowly, deliberately, and if possible only on the areas of interest.

Do this diligently and the bottlenecks will become clear. It's a slow process and not particularly exciting, but it is the first sure step on the way to completion.

Code cleanup

ONCE THE BOTTLENECKS HAVE BEEN FOUND, the next task is to clarify and simplify the bottlenecked code.

The very first step is to check if any unit tests exist for the bottlenecked code. If not, place a call to the software quality assurance team and have them put together a proper test suite for it. It is important that one's efforts at code optimization not introduce regressions.

Once a test suite exists and the bottleneck passes them, the next step is to clarify and simplify. Rewrite the bottleneck in a manner that a talented college undergraduate could easily follow along. At this point it does not matter if the changes have a negative impact on performance. The important thing here is to have a clean and simple base of code from which to begin one's efforts. The process of simplification will also tend to reveal the algorithmic structure of the bottleneck. What precisely is it doing? What inputs are used, and how? This information is of critical importance to the next step.

Do not proceed further until the code has been clarified, the test suite reports no defects, and profiling data has been regenerated using the new codebase. After all, it is possible the clarification has opened up the bottleneck. Without current profiling data it is impossible to know.

Algorithmic performance

NEVER OPTIMIZE CODE when the algorithm itself can be optimized. Perhaps every now and again the input data set is already ordered and that causes the `quicksort` routine to go into worst-case performance. One option would be to rewrite the `quicksort` routine in Assembler for more speed. Another far better option would be to add a couple of lines of code to randomize the input set. Be careful about hand-rolling Assembly to make a naïve matrix multiplication algorithm go faster: look into Strassen's algorithm.

It is not necessary to have an encyclopedic knowledge of algorithms. The field of computer science is filled with journals, textbooks and monographs on algorithms. Develop research skills. Learn how to describe the algorithm used by the code, learn how to look for it in the literature, and learn how to springboard from that to finding other and better algorithms.[3]

[3] A good friend once had a simple programming problem he couldn't optimize. After doing research and talking to some algorithm people, he discovered his 'simple problem' was NP-Complete and it was not possible to optimize it for speed. Sometimes research shows us how to achieve breathtaking improvements. Just as valuably, research can also show us no improvements are possible.

It's tempting to think the problem is so novel that a literature survey will be fruitless. This line of thinking is unfortunately commonplace in the industry. It is also almost always wrong. Somewhere in the world, some academic has spent a great deal of time thinking about the most efficient way to solve the problem at hand. It is the height of foolishness to not even look for this person's results.

Re-profiling

NOW THAT THE CODE IS SIMPLIFIED , the literature surveyed, and the code re-profiled, it is time to start opening up the code.

The technique used will vary according to programming language, operating system, operating environment and more. As a general rule, though, write the bottleneck in a language closer to "bare metal." If the codebase is in Python, implement the bottlenecked code in C. If the codebase is in C, implement the bottlenecked code in Assembly, and so on and so on.

Avoid using clever tricks to fool the compiler into acting a certain way. These tricks are non-portable and often change between compiler versions. If the compiler is not generating efficient-enough code and there is no way to clearly direct the compiler to change its code generation, then eschew the compiler altogether. Do not rely on undocumented side effects, implementation details, or anything that can be called "compiler weirdness."

Remember that a compiler is an expert system for generating machine code. Like all expert systems, a skilled human being can beat it in a small enough challenge. Very few programmers are capable of beating a compiler for more than a few hundred lines of Assembly, though, and a wise programmer keeps this in mind.

Closing remarks

THE METHODOLOGY PRESENTED HERE is simple, direct, and rather boring. For all of that, though, it works astonishingly well.

That said, do not fall into the trap of thinking this methodology will always work. It won't. It is useful to probably ninety percent of optimization problems, though, and for that reason I believe this methodology should be part of every programmer's repertoire.