A Case Study of Performance Tuning

Michael R. Dunlavey
Certara/Pharsight Corp.
MikeDunlavey44@gmail.com

Abstract

Experience shows that large software, as written, can contain multiple performance problems. Taken together, these can comprise a large fraction of execution time, close to 100%, such that their removal can give orders of magnitude speedup. However, to achieve this speedup, it is important that no such problem be missed.

This paper describes an old but little-known method for locating performance problems, informally called "random pausing". The statistical properties of sampling are explored, including *false positives* (non-problems found), and *false negatives* (real problems missed). We provide a concrete example in which a series of problems are identified and fixed, resulting in a speedup factor of 730. The problems that would be missed in that example, without using the present method, are explained.

1. Introduction

1.1 The Problem

Figure 1 illustrates a hypothetical program taking some length of time (*100* sec, say). It contains six performance problems[1], labeled A through F, taking percentages of overall time of *30%*, *21%*, etc. If those six problems are removed, time is reduced to *11.8* sec. The speedup factor is *100/11.8 = 8.5*. However, if one of those problems is not removed, such as D, the speedup factor is *100/(11.8+10.3) = 4.5*. That speed differential, *4.5* vs. *8.5*, exemplifies the price paid for false negatives, and the reason they should be avoided.[2]

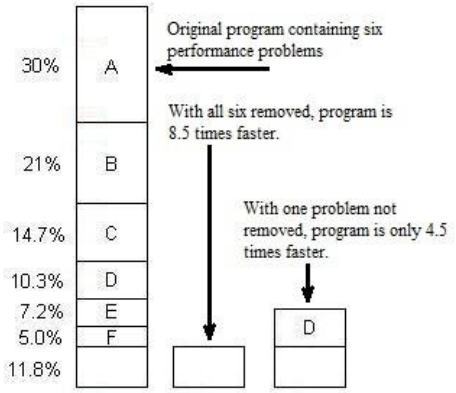

Figure 1. How multiple performance problems can exist within a program, and why they should all be removed.

1.2 The Method

The method used is sampling of the program state. However, rather than taking a large number of samples programmatically, and summarizing them statistically, we take a small number manually and apply the programmer's attention to each one. Since the programmer can examine the full state of the program in a sample, such as each line of code on the call stack, and relevant data context, the complete reason for the activity can be understood. If it is seen on multiple samples, and if there is a faster way to accomplish that purpose, a performance problem has been identified.

When this is compared to the use of statistical summaries of, say, fraction of execution time used by functions or individual lines of code, those summaries may give greater statistical precision of timing, but they are necessarily vague about the contextual information identifying the reasons for the execution. Furthermore, we will show how a small number of samples gives entirely adequate measurement precision to satisfy the goal of identifying problems.

2. Justification

2.1 Predicates

Whenever a sample is taken, it is described by some kind of predicate. One form of predicate is "program counter is in function F". Summarizing the fraction of samples satisfying that predicate is "self time". Another predicate is "function F is on the stack". Summarizing that predicate gives "inclusive time by function". Another is "function F is above function G on the stack". That gives the fraction of time that F is in the process of calling G, the kind of information given in call graphs. These are the kinds of predicates summarized in typical profilers.

Another kind of predicate is much more application specific, or "semantic", such as "the program is in the process of reading a dll file so as to extract a string resource

[1] We use the term "problem", although a better term might be "opportunity", because often the code is perfectly defensible. It's just that it takes a large fraction of time, and there's a way to reduce it. Occasionally these problems are algorithmic in the "big-O" sense. More often they are simply a matter of constant factors.

[2] If removal of a problem saves fraction *X* of the time, but it originally consumes fraction *FX* of the time (*F > 1*), that makes it easier to find, compared to what it saves. For simplicity of exposition, and since problems are often easily fixed, we assume throughout that *F=1*. Given the broad uncertainties in measurement, it does not significantly affect the argument.

for the purpose of internationalizing the UI, but the string being extracted is not actually used in the UI, or it is used in the UI but is not likely to be internationalized". Predicates that describe performance problems are not all of the form that can be easily discovered by automatic statistical summaries. Nevertheless they can exist, and if one is not found during performance tuning, it is a false negative.

2.2 Statistics of Sampling

If a given problem, when removed, would save fraction *X* of execution time, then a random sample of the program's state has probability at least *X* of occurring during that time. So each sample can be considered as a simple coin-toss Bernoulli trial, and if *N* samples are taken, the number of samples that occur in the problem, or "hit" it, is *S*, described by a Binomial distribution[1]. So we can write

$$S \sim Binomial(N, X)$$

The mean of *S* is *NX*, and its variance is *NX(1-X)*. So, for example, if *X=0.3*, and *20* samples are taken, the number of them that show the problem has a mean of *NX=20*0.3 = 6*, and a variance of *NX(1-X) = 4.2*. The standard deviation is the square root of that, or *2.05* samples.

Consider the reverse problem. If we take *N* samples, and we see something we could remove on *S* of them, what does that tell us about the fraction *X* of time it takes? The distribution that answers that question is *Beta*, as in

$$X \sim Beta(S+1, (N-S)+1)^3$$

That distribution has its mode (most likely value) at *S/N*, not surprisingly. If *N=20*, and *S=6*, the most likely value of *X* is *0.3*. The mean of the distribution is closer to the center; it is *(S+1)/(N+2) = 7/22 = 0.318*.

The number of samples needed depends on prior knowledge. A special case is if the program is taking far longer than we know it should, or has an infinite loop. That says *X* is close to *1.0*, and a single sample is nearly certain to show the problem.[4] Without that prior knowledge, a single sample doesn't say much. However, if a problem appears on more than one sample, then we can be sure removing it will give a significant speedup. The fewer samples taken before seeing the problem twice, the larger *X* is likely to be. Figure 2 shows the distribution of *X* if *N* samples are taken and the problem appears on *S=2* of them.

An issue that often concerns people is a *false positive*, or having the method identify a problem for which *X* turns out to be disappointingly small. As Figure 2 shows, while there is a probability that *X* will be disappointingly less than its mode or mean, there is an equal probability that it will be gratifyingly larger. So it is a gamble, but the odds of significant speedup are quite good. Regardless, if greater certainty is desired, more samples can be taken.

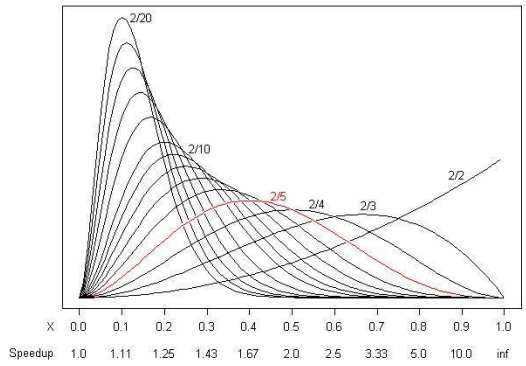

Figure 2. *Beta* distribution of problem cost *X* as a function of the number of samples *N*, given that it is seen two times (*S=2*).

The actual speedup factor follows a different distribution. If fraction *X* is saved, the program speeds up by a factor *Y = 1/(1-X)*[3]. So, for example, if *X = 0.3*, then *Y = 1/(1-0.3) = 1.43*. The distribution of *Y* is *BetaPrime*:

$$Y \sim BetaPrime(S+1, (N-S)+1) + 1$$

for which the average is *(S+1)/(N-S) + 1*, as shown in Figure 3. For example, if *N=3* samples are taken, and the problem is seen on *S=2* of them, the average cost *X* is *(2+1)/(3+2)=0.6*, but the average speedup factor is not *1/(1-0.6)=2.5*, it is *(2+1)/(3-2)+1=4*, because of the right-skewing of the distribution. This becomes very pronounced as *S/N* increases, going to "infinity" at *S=N*.

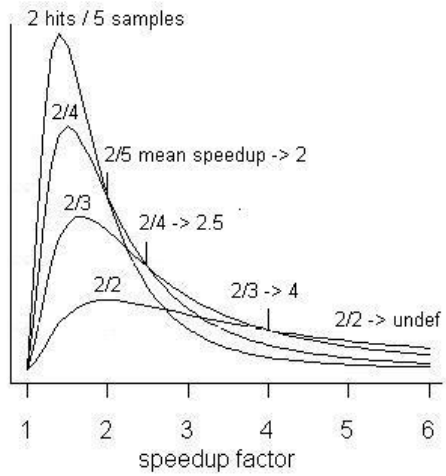

Figure 3. Distribution of speedup factor *Y* as a function of *S*, *N*.

Another useful question is, given the cost of the problem *X*, how many samples *N* must be taken in order to see the problem two times? For that, *N* has a *Negative Binomial* distribution:

$$N \sim NegativeBinomial(2, X) + 2$$

[3] Assuming a flat Bayesian prior of Beta(1,1)[2].

[4] It could be that an infinite loop is caused by multiple problems, such as A, B, and C. In that case, samples will identify at least one of them which, after removal, produces an infinite loop running faster. Then the others can be removed the same way.

and the mean is *2/X*. For example, if *X* is *0.3*, it takes an average of *6.67* samples to see the problem twice.

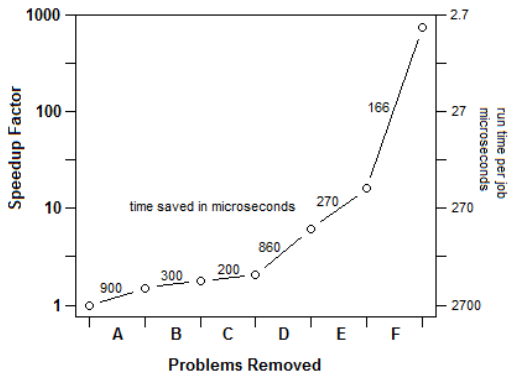

Figure 4. Summary of the multi-stage process of performance tuning the example program.

3. A Case Study of Performance Tuning

Figure 4 illustrates the process of performance tuning an example program, in six stages, resulting in an overall speedup factor of 730. This is to illustrate the importance of not getting any false negatives. Any of these six problems not fixed would have a severe impact on the final speedup factor.

The program is a discrete simulation of a factory floor environment[4], in which there are units of work that progress through a succession of workstations and have operations performed on them. The main performance measurement is the number of microseconds needed to simulate a unit of work. This is the measurement that is reduced by a factor of 730.

3.1 First Iteration

Five samples are taken. Two of them are in `new` and `delete`. The other three are in methods on `std::vector`, two in `push_back` and one in `size`. (Problem A.) This suggests trying a different vector class. Result: time decreases from 2700 to 1800.

3.2 Second Iteration

Ten samples are taken, of which four were in the array indexing methods `[]`. (Problem B.) This was replaced by direct indexing, reducing the time to 1500. It is possible that if the samples were taken on a release build, this problem would not have existed because the indexing would have been direct. The problem with that is a release build, even if it contains symbolic information, makes it harder to interpret a sample, causing problems to be missed and the speedup progression to stall. (This is why, in our opinion, the time to use compiler optimization is at the end of the process, not the beginning.)

3.3 Third and Fourth Iterations

Ten samples are taken. Adding an item to an array applied to two of them. An item destructor applied to two of them. `new` applied to two of them, and `RemoveAt` applied to one of them.

Two changes were done. First, the arrays were replaced with linked lists, reducing the time to 1300. (Problem C.) Then to reduce the time spent in `new` and `delete`, used objects were pooled for re-use, reducing the time to 440. (Problem D.)

This is an example where the problem was very non-local. Profilers that summarize by local code positions would have missed the larger points, that in this case, linked lists could have faster add/remove time, and enough time was spent in allocating/deleting objects to make recycling them worthwhile.

3.4 Fifth Iteration

In the third iteration, part of the change from arrays to linked lists was to introduce a macro called `NTH`, to use for getting the *n*th element from a linked list. This macro was now using a large percent of the time, so it was replaced by directly pointing into the list. (Problem E.) This reduced the time to 170.

Profilers that summarize by function would not have pinpointed that macro. The ones that summarize by line of code might or might not have seen it, depending on how they handle macros.

3.5 Sixth Iteration

Now four samples were taken, and every one was in the process of doing I/O to record completion of a job to the console. (Problem F.) Since that was not really necessary, it was commented out, reducing the time to 3.7.

Profilers that look only at CPU time, rather than wall-clock time, would not have seen this. The ones that summarize by function might have left the user wondering the reason for the I/O, if there were multiple I/O statements in the program.

4. Related Work

Any discussion of profiling must reference *gprof*[5], which was created to try to address the shortcomings of preexisting profilers that only sampled the program counter and reported which functions contained it ("self time"). It did this by counting the number of times any function `A` called any function `B`, and then attempting to "charge back" the low-level self-time to higher level routines. It had a number of difficulties, including looking only at CPU time and having trouble with recursion. A larger issue is that expectations have been placed on it that were never actually claimed, namely that it was effective for *locating* (as opposed to *measuring*) performance problems.

More recent profilers, such as *Zoom*[6], sample the stack, not just the program counter. They sample on wall-clock time, so they can see problems that are non-computational, such as needless I/O. They report at the line-of-code level, not just functions, so they are far more precise in problem location. Recursion is not an issue, because the percent of time a function or line of code is responsible for is simply the percent of time it is on the

stack, regardless of how many occurrences of it are on a stack sample.[5]

Slowdowns due to competition with other processes are not much of an issue because they do not strongly bias the statistic that most directly indicates problems - the time-percentages of lines of code or functions. In spite of all these improvements, there are problems they do not easily identify, false negatives, because they cannot include enough contextual information to determine the need for the time being spent, and thus identify semantic problems.

One key difference between the work presented here and research on traditional performance profiling regards the assumed need for measurement precision[7,8]. Here we show that, while a larger number of samples can reduce the probability of false positives by means of smaller measurement variance, if it is achieved by applying less scrutiny to individual samples, it can result in false negatives, preventing the achievement of higher speedup factors.

The notion of random pausing has been previously explored by Dunlavey[9,10].

5. Discussion

One weakness of the present method is it relies on being able to discern the full reason why a given moment of time is being spent. The call stack and data context is usually rich in such information. However, if a program involves communication protocols across multiple independent processes, or if the program works in "message-driven style" where it may be difficult to discern the reason why some work has been requested, then the present method is less effective. For protocol-driven programs, we have used a logging technique that is laborious but effective. For message-driven programs, they would be easier to optimize if messages were modified so as to provide some sort of back-trace information.

One possible objection to the method is this: Suppose the program has only small problems. Then the method would have a hard time finding them. We consider a small problem to be one for which $X = 0.05$ or less. So the average number of samples N needed to see it twice would be $2/0.05 = 40$ samples, a somewhat large number. Certainly it is possible to construct such a program, by taking an otherwise fully optimal program and de-tuning it by *5%*. Nevertheless, while such programs exist, one cannot assume that any given real program, not constructed that way, is among them. To prove that at least some programs are not like that is the reason we've given this case study. Our experience gives two points: 1) Often programs that are supposedly nearly optimal actually have sizeable problems whose presence is not suspected. 2) In some highly optimized programs, such as hardware-level graphics programs, the samples show that indeed nearly all the time is spent in the highly optimized loops. However, what this means is, in addition to taking more samples to find small speedup opportunities, a place to look for substantial benefit is to try to further optimize those loops, by unrolling, being cache-conscious, etc.

Another objection says the random pausing method would work on small programs but not large. That is the opposite of our experience. Larger software tends to produce deeper stack samples, with more function calls. If any such call appears on more than one sample and can be eliminated, a significant speedup results, so deeper stacks tend to mean "better hunting". Another way to look at it is, suppose more calls are made at each level than are strictly necessary, by some small factor whose geometric mean is R, like 1.1. Then if there are M (like 30) stack levels on average over time, that produces an exponential slowdown of R^M. For example 1.1^{30} gives a slowdown factor of over 17. What's more, the slowdown is *diffuse* – there is no particular "slow method" to be found.

A broader issue is the supposed tradeoff between performance and maintainability, regardless of the method of finding performance problems. In our experience, the changes made to fix them need not affect maintainability, provided they are accompanied by commentary. For example, saving the result of a function call in a variable, rather than calling the function multiple times with the same arguments, is not hard to explain.

[5] In case this is not obvious, suppose samples are taken every 10 milliseconds, for a total of N samples, and X is the fraction of them showing the recursive function call. If that call were somehow made to take no time, such as by deleting it, jumping around it, or passing it off to an ultra-fast processor, the topmost call would have no exposure to being sampled. So those samples would disappear, shortening the total time by fraction X.

Acknowledgements

Jon Bentley was helpful in providing feedback on a draft of the paper.

References

1. **Evans, M., Hastings, N., Peacock, B.** *Statistical Distributions, 2nd Edition.* New York : Wiley, 1993. ISBN 0-471-55951-2.

2. **Lee, P.** *Bayesian Statistics, An Introduction, 2nd Edition.* London : Arnold, 1997. ISBN 0-340-67785-6.

3. **Amdahl, G.** Amdahl's Law, http://en.wikipedia.org/wiki/Amdahl's_law.

4. **Dunlavey, M.** Random Pause Demo, http://sourceforge.net/projects/randompausedemo.

5. **Graham, S., Kessler, P., McKusick, M.** gprof: a Call Graph Execution Profiler. *ACM SIGPLAN Notices.* 1982.

6. **RotateRight.** Zoom, http://www.rotateright.com.

7. **Ponder, C., Fateman, R.** Inaccuracies in program profilers. Software Practice & Experience 1988, Vol. 18(5), pp. 459-467.

8. **Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.** Evaluating the Accuracy of Java Profilers. *PLDI '10.* 2010.

9. **Dunlavey, M.** Performance tuning with instruction-level cost derived from call-stack sampling. *ACM SIGPLAN NOTICES.* August 2007, Vol. 42, 8, pp. 4-8.

10. —. *Building Better Applications: a Theory of Efficient Software Development.* New York : International Thompson Publishing, 1994. ISBN 0-442-01740-5.