Performance Tuning with Instruction-Level Cost Derived from Call-Stack Sampling

Michael Dunlavey
Pharsight Corporation
276 Harris Ave., Needham, MA 02492, USA
mdunlavey@pharsight.com

Abstract

Except for program-counter histogramming, most modern profiling tools summarize at the level of entire functions or basic blocks, with or without additional information such as calling context or call graphs. This paper explicates the value of information about the cost of specific instructions, relative to summaries that do not include it. A good source of this information is time-random sampling of the call stack. To get the diagnostic benefit of instruction costs it is not necessary to measure them with high precision or efficiency. In fact, manual sampling suffices quite well, when it can be used. Other benefits of call stack sampling are that it can be used with unmodified software and libraries, and it is easily confined to the time intervals of interest. As with other profiling techniques, it can be employed repeatedly to remove all significant performance problems in single-thread programs.

Keywords: Optimization, bottleneck, performance analysis, software development, profiling, instrumentation, call stack sampling.

1. Introduction and Related Work

In single-thread software, the cost of any individual instruction, whether a function-call instruction or not, can be defined as the total amount of time it is on the call stack during the time interval of interest, because if it could be somehow eliminated or made to take no time, that is how much would be saved. The cost of any set of instructions within a single function is just the sum of the costs of the individual instructions.

When a performance problem has been definitively identified within a function, fixing it requires replacing a certain set of instructions A with another set B. The time saved is the cost of A minus the cost of B. Therefore, the cost of instructions is very useful in finding performance problems.

A very simple way to estimate instruction cost is via call stack sampling. It has been demonstrated, when used on large applications, to achieve speedup factors of more than 10 during execution intervals of interest[Dunlavey]. This is not surprising, considering the experience that in large software there can often be several independent performance problems consuming most of the original execution time.

To get the diagnostic benefit of call stack sampling, neither accuracy nor efficiency are essential. One method is to obtain a small number of time-random samples of the call stack, typically from 5 to 20, during the execution period of interest, over one or multiple runs. As each sample is collected, it is examined for call site addresses or terminal addresses that have appeared on previous samples. The code at such an address is examined to see if a way could be found to replace it with something faster, because the fact that it appeared on multiple samples means it very likely has a significant time cost. If the code cannot be replaced, the sampling continues. If it can be replaced, the program is rebuilt and the entire process is repeated to find the next performance problem. Depending on the software, it may be useful to examine additional context at each sample, such as the data structure being operated on, or the stack of callers. Knowing that significant time is being spent in that code, the object is to find out *why* it is there and ask if something faster could replace it.

The method finds both call sites and "hot spots", regions of code that consume significant time without calling functions. While large applications sometimes have hot spots, experience shows that most problems manifest at call sites[Dunlavey].

Samples can be collected in a profiling tool, or manually, with a "pause" key in a debugger. If in a profiling tool, the summary statistic should be, for each terminal address or call site address, the fraction of call stacks on which that address appears, because that directly estimates the instruction's cost, no matter what its execution count is or time per invocation.

No matter which profiling method is used, it is worth mentioning that the iterative process of find-and-remove has an interesting property. If any two performance problems are independent of one another, as they often are, removing one of them means the other takes a larger percentage of the remaining time. This "magnification effect" allows the process to continue until all significant problems that can be removed have been. As the execution time grows short, it may be useful to insert a temporary outer loop to amplify the remaining problems.

In the literature, performance tuning of large software generally depends on one or both of two approaches, instrumentation and call stack sampling, and tends to summarize at the level of functions or pathways.

[Ammons97] shows how to use hardware counters to collect function call-tree statistics of various resources, and deemphasizes information about static code elements such as single procedures or statements. In this paper 1) we assume the essential resource is wall clock time, and 2) the only code one can edit is just those fine-grained static code elements, i.e. statements, so it is important to know what they cost. We would agree that contextual information is important, but not because it helps find the costly code. Samples tell us exactly where the costly code is. The value of context is that it helps answer *why* that code is being executed. Code runs for a purpose, and if we know the reason we may be able to find faster code to accomplish the same purpose.

[Ammons04] tells of the BOTTLENECKS system which collects information from various tools, whether instrumentation or call stack sampling, and forms them into call trees. These call trees are heuristically examined for possible performance problems, and those are precisely estimated and classified. Among the design issues is to minimize reporting of redundant (overlapping) bottlenecks. We would suggest that the answer is simple: only fix one or two at a time. In one application, a performance gain of 23% is reported. There might have been other bottlenecks. If there were, the removal of the first set would have magnified the remainder by 30%, making them easier to find.

The system called **`csprof`** also forms call tree summaries, but using input from call stack samples[Froyd05]. In fact, **`csprof`** retains call-site information, but it does not seem to summarize at that level.

There are many profiler tools that perform instrumentation, most notably the original **`gprof`**[Graham04]. Some commercial tools perform call stack sampling [Sun, SGI]. In nearly all of these profilers, the form of summary is either "flat", "call graph", or "call tree". In the flat summary, for each function, the number of calls and time spent in/under the function are presented. In the call graph summary, this information is broken down by immediate callers and callees. In the call tree summary, the information is subdivided for each function at a node in the call tree. In most tools these summaries either don't have, or have but discard, the key piece of information, namely the time cost attributed to individual call sites or terminal instructions.

Some tools that do display the crucial instruction-level cost are [Sun, SGI].

Without instruction-level cost, if a developer attempts to optimize a costly function, he/she must use intuition to locate the instructions/statements within it that generate the cost. This can be a problem if the function is large or complex. Call graph, call tree, or basic block information can narrow down the search somewhat, but not eliminate it in general. When instrumentation is being used, calls to uninstrumented functions, such as library functions, may not even appear. Examining the source code will not show function calls that are inserted by the compiler. However, if call site information is retained, the costly call sites are identified, eliminating guesswork. It is commonly accepted that programmers are poor at guessing where performance problems are.

This paper deemphasizes certain established goals of profiling – accuracy and efficiency. Once a potential problem has been identified via call stack samples, one can rest assured that it costs enough to consider removing, even if the cost is not known to multiple decimal places. If more precision is desired, additional samples may be taken. If there are enough samples, the confidence intervals of cost can be derived using the normal approximation to the binomial distribution. However, the exact time saved by removing the problem can always be measured by post-hoc timing.

Any discussion of efficiency of performance tuning must include the human time spent preparing the software and analyzing the results, and must hold foremost the goal of finding performance problems and getting the benefit of their removal. If samples are collected automatically, efficiency is not an issue because the number of samples need not be large. With manual call-stack sampling and collection of instruction-level cost, there is little preparation, and analysis happens at the same time as samples are collected, when all context is available. Efficiency is not an issue since the program is stopped anyway. In fact, in the author's experience, even if it is necessary to decipher memory dump listings to read the call stack, the information is worth it. When the business cost of poor performance is large, convenience for the programmer is nice but is not the driving concern.

Common Types of Performance Problems

It is illustrative to list some of the kinds of performance problems that have been found via call stack sampling, in the author's experience.

Algorithm choice. A simple $O(N^2)$ sort of a table of strings could be adequate until the size of the table gets larger than anticipated. For example, samples

could show the PC (program counter) in a string compare function, with the statement

```
if (stringA > stringB)…
```

higher up on the call stack. This clearly indicates the problem, and the algorithm can be replaced with an O(NlogN) sort, eliminating most of the cost.

Lack of caching. Sometimes there is a function that collects information from some source and packages it nicely for use, like

```
myObject = GetStuffFromDB();
```

This may be so convenient to call that it is called more often than necessary, when code to cache the prior result might be inconvenient to write. If this results in significant time being used, it quickly shows up in the samples.

Another example of this is the following:

```
for(i=0; i<myList.Count; i++)…
```

If the list is short, it may not matter, but if the list is long, or if it costs more to determine its length than anticipated, this could show up on the samples. Of course, one possible fix is to store the length in a variable and use that.

Using a too-general data structure. Sometimes, due to concerns about generality and coding style, feature-laden collection classes may be used, with the result that significant time is spent in creating, destroying, and using these objects. If this results in a performance problem, samples will show it. Often the fix is to use simple arrays instead.

Using a too-powerful function. Sometimes a function that calculates a number of things is called when only one of its results is actually needed. The rest are discarded. If this uses significant time, the samples show the function being called in that way.

Using too-local objects. Sometimes the sample is in some memory allocation routine or construction code of an object. Higher up the call stack, one could see:

```
while(…){MyClass tempOb(); …
```

In other words, the creation and initializing of the object is taking significant time. Simply moving the object declaration out of the loop could fix the problem. This also removes time spent in deallocation or collection of the object.

Being "I/O bound". It may be that a program is considered I/O bound because it spends all of it's time doing `printf` or `scanf`. However, often the call stack samples show that a lot of the time is actually going into formatting – converting between textual and binary forms of numbers, for example. If the I/O is to a file not meant for human eyes, one could consider writing or reading binary. A related type of problem can occur when accessing a database, such as opening and closing it more often than necessary.

Slavish adherence to design. An example of this is a system with two modules, A and B, communicating with each other via XML strings. One could find samples in the routines for writing and parsing XML, and constructing and destructing attendant data structures. This might just be considered a cost of doing business. On the other hand, a simpler format might be used. It could even be that modules A and B are in the same memory space and could simply share data. (This really happened.)

Cache misses in inner loops. These occur in inner loops that go "against the grain" of local memory structure. They manifest as samples in which the program counter is in that code, where it can be examined for non-locality.

The remainder of this paper is devoted to two examples that illustrate the method of call stack sampling with small numbers of samples and retention of call site information. The method is compared to summary techniques that do not use instruction-level information.

Simple Example 1

An $O(N^2)$ sort algorithm on an array of numbers is embedded somewhere in a million-line application. When exercised on one set of numbers, it takes one second to complete and is not noticed. When exercised on a larger set of numbers, it takes 100 seconds to complete and appears to hang the program. If it is interrupted at a random time in those 100 seconds, the PC (program counter) will, with near certainty, be in the inner loop of the insertion sort code, informing the observer of the location of the hot spot. That is *one* sample.

Next, the program is changed to sort a set of strings, using a string compare function. Now, when it is halted, the PC is highly likely to be found in the string compare function. However, one level up on the call stack is the call to that function, and it is in the insertion sort code, informing the observer of the location of the problem.

This example demonstrates that the more costly a problem is, the fewer samples are needed to find it. It also demonstrates that with instruction-level information, the problem is located without guesswork. It is only necessary to ask, at each level, if that code could be optimized.

Simple Example 2

Following is a highly artificial program designed to illustrate the technique. It is a distillation of many real projects from the author's experience. To extrapolate to typical real software, multiply the number and size of functions by factors of 10 to 1000, and assume staff turnover so no one person understands all of it.

Programmers do not, as a rule, intentionally write code that they know to be unnecessary. However, once they find out that a particular statement is costing a lot of time, they will examine it carefully and may discover that it could be replaced, and in that sense the statement is not really necessary. So, in this program, we mark statements as *necessary* or *unnecessary*, indicating what the programmer will discover when they find out they should look there.

```
 1 void main(){
 2     A(); /*necessary*/
 3     A(); /*unnecessary*/
 4 }
 5 void A(){
 6     B(); /*necessary*/
 7     B(); /*necessary*/
 8 }
 9 void B(){
10     C(); /*necessary*/
11     C(); /*unnecessary*/
12 }
13 void C(){
14 /*necessary 1-sec hotspot */
15 }
```

The process of analyzing this program is compared, first using shallow (program counter alone) sampling, then using function timing instrumentation, then using call graph timing, then using call stack sampling.

Using PC Sampling

A statistical sampling of the program counter yields the following information:

Line of Code	% of time active
14	100

This causes the programmer to look at line 14. It is a hot spot, but since it is necessary, nothing can be done.

Using Function Timing Instrumentation

Function instrumentation yields the following information:

Function	# Calls	Average call duration	Total time in/under function
main	1	8 sec	8 sec
A	2	4	8
B	4	2	8
C	8	1	8

To make sense of this, the programmer needs to have some a-priori knowledge of approximately how many calls of each of these functions is considered normal. They all consume 100% of the time, so they are all about equally under suspicion. In other words, educated guesswork is needed. This is difficult when the program is scaled up to realistic size and there is staff turnover.

Using Call Graph Instrumentation

A call graph consists of a set of arcs, each of which is a pair of functions, the first of which contains calls to the second. Summaries of these calls can be accumulated:

Caller	Callee	# times	# times total	Total duration
main	A	2	2	8 sec
A	B	2	4	8
B	C	2	8	8

Again, it is unclear how to make sense of this without educated guesswork which is problematic as the program is scaled up.

Using Call Stack Sampling

In this artificial case, we assume samples of the call stack are taken at specific pseudo-random times (in the middle of every second), and at each time the call stack is recorded

Sample	Lines on call stack
1	2, 6, 10, 14
2	2, 6, 11, 14
3	2, 7, 10, 14
4	2, 7, 11, 14
5	3, 6, 10, 14
6	3, 6, 11, 14
7	3, 7, 10, 14
8	3, 7, 11, 14

From this, the percentage of call stacks on which each line appears is easily calculated:

Line	On % of call stacks

2	50
3	50
6	50
7	50
10	50
11	50
14	100

First Iteration of deep sampling

- The most frequently seen line is 14. It is examined. It is a hot spot, but since it is necessary it is ignored.
- It is seen that line 10 appears on 50% of call stacks, but on examination, it is necessary, so it is ignored.
- Line 11 also appears on 50% of call stacks. It is looked at and found to be unnecessary, so it is removed.

Result: Execution time is reduced by 50% to 4 seconds.

Second Iteration

- On the second iteration, line 3 is seen as being on 50% of the call stacks, and since it is seen to be unnecessary, it is removed.

Result: Execution time is reduced by 50% to 2 seconds.

Third Iteration

- On the third iteration, various lines are seen as being on large fractions of the call stacks, but since they are all seen to be necessary, the process stops.

Overall Result: Execution time is reduced by a factor of 4.

Conclusions

While instruction-level cost is summarized in some tools, it has generally not been known that it is strongly diagnostic of performance problems. Time-random sampling of the call stack is a good way to get that information. Accuracy and efficiency of estimating cost is not essential. Manual sampling can be quite effective. Instruction-level cost has been used to aggressively tune the performance of large software.

Acknowledgements

Thanks to Bob Leary for useful feedback and insights.

References

[Ammons97] Glenn Ammons, Thomas Ball, James Larus, Exploiting hardware performance counters with flow and context sensitive profiling, ACM SIGPLAN, PLDI-97.

[Ammons04] Glenn Ammons, Jong-Deok Choi, Manish Gupta, Nikhil Swamy, Finding and Removing Performance Bottlenecks in Large Systems, European conference on object-oriented programming, Oslo, Norway, 2004, http://pages.cs.wisc.edu/~ammons/bottlenecks.pdf

[Dunlavey] Michael Dunlavey, Performance Tuning: Slugging It Out!, Dr. Dobb's Journal, Vol 18, #12, November 1993, pp 18-26. Also: Building Better Applications: a Theory of Efficient Software Development, International Thomson Publications, NY 1994, ISBN 0442017405

[Froyd05] Nathan Froyd, John Mellor-Crummey, Rob Fowler, Low-Overhead Call Path Profiling of Unmodified, Optimized Code, Proceedings, 19th annual conference on Supercomputing, 2005.

[Graham04] Susan Graham, Peter Kessler, Marshall McKusick, gprof: a Call Graph Execution Profiler, ACM SIGPLAN Notices, Vol. 39, #4, April 2004, pp 49-57

[SGI] SGI Altix Applications Development and Optimization, http://sc.tamu.edu/help/SGI.Tutorial/sgi-tutorial.pdf

[Sun] Sun Studio Performance Analyzer. http://developers.sun.com/sunstudio/analyzer_index.html