Multiplicative Speedup of a Moderate-Size Program in Multiple Stages

Michael R. Dunlavey, *Certara/Pharsight Corp.*

Abstract

Profilers find speedups, but it cannot be assumed they find them all, or that the missing ones don't matter. An example is given of a moderately large program, containing several speedup opportunities, with a distribution of sizes, that together account for nearly all of the execution time. Effecting all of them results in a large total speedup factor, but omitting any results in far less. Explanation is given why some of these speedups would not easily be found using traditional summarizing profilers, but they are found by an ancient but little-known manual method that looks at a small number of samples. The statistical justification for the method is explained.

1. Introduction

Ammons, et al [Ammons] in 1997 mentioned two central issues with profilers, simple metrics applied to static constructs like procedures or statements, and aggressive reduction of information.[1] [Bentley] has emphasized the means and importance of squeezing constant factors in performance tuning. This paper gives an example of a moderately complex program being aggressively accelerated by de-emphasizing metrics and by maximizing use of available information.

In this example, reasonably well-written code, with good "big-O" performance, can contain a great deal of room for constant factor speedup, in ways that do not much hurt maintainability. They exist only because it was difficult to foresee, when the code was written, what all the appropriate areas for speedup would be.

While this paper shows some useful techniques to get speed, *that is not the point.* The real issue is *finding the speedups*. If they are fixed before they are found, it is a case of "ready-fire-aim".

There are different kinds of performance profilers. Rather than survey them, we suggest the best are the stack-samplers, with instruction-level granularity, on wall-clock time[Zoom]. Wall-clock time is important because it cannot be assumed, in large software, that the programmer knows about and approves of all the voluntary blocking done by a process. In highly multi-threaded systems, this still applies to the threads of interest.

The problem with even the best profilers is the summarizing back-end. Even though each sample contains full information about why time is spent, that information requires an intelligent agent to understand it, in the context of what the code is trying to do. Simply summarizing it, even with call graphs, hot paths, etc., loses much of that information. The result is unrecognized opportunities for speedup.

Every speedup opportunity has a size, as a fraction of wall-clock running time, and that is roughly the probability it will be seen on any given sample. When it is seen on *more than one sample*, its size is very roughly known, as explained below, but the *identity* of the speedup is precisely known. (Infinite or long-running loops only need one sample.)

1.1. Descriptions of Samples

If any statement is on the stack for fraction *X* of the time (even with recursion), it is responsible for that time fraction, regardless of the number of times it is executed. If the statement can be made to take no time (such as by removing it), total execution time will shorten by fraction *X. The same goes for any description one cares to make*, such as "function F is on the stack", or "function F is calling function G from line L", or "function F is being called with the same arguments as it has been called with previously", or "the program is currently in the process of allocating memory in order to grow a particular data structure", or "the program is currently reading a file for a purpose that, ten levels up the call stack, could easily be avoided". The latter descriptions can only be formulated by a human programmer examining a stack sample.

1.2. Objections

An objection is that a human cannot examine enough samples. However, speedups of large size do not need many samples to be seen twice, and when they are effected, the size of smaller ones is amplified by the speedup factor. This is how a series of speedups can be found, as explained below.

Another objection is that examination of a small number of samples will identify disappointingly small speedups (a so-called "false positive" in statistical language). However, the probability is small, as explained

[1] "Most profiling systems suffer from two major deficiencies: first, they only apportion simple metrics, such as execution frequency or elapsed time to static, syntactic units, such as procedures or statements; second, they aggressively reduce the volume of information collected and reported, although aggregation can hide striking differences in program behavior."

below, and is always accompanied by an equally likely performance jackpot.

Another objection is that competition from other processes will skew the results. That has the same effect as running on a slower CPU, or under a simulator; it does not affect time fraction seriously enough to be an issue.

2. Statistics of Sampling

It is not necessary to know the statistics of sampling to do profiling. It is included here only to explain why a small number of samples is sufficient for finding reasonable-size speedups.

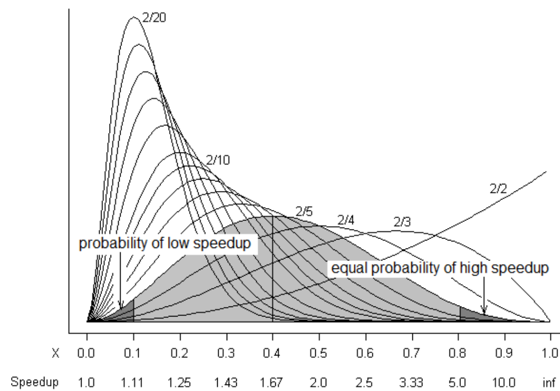

Figure 1. Distribution of X, given *s/n*.

Figure 1 shows the distribution of X given that s out of n samples exhibit a common description. Technically we say $X \sim Beta(s+1, n-s+1)$[Lee, Evans]. The shaded area is for $s/n = 2/5$, and the darker areas show the probability of getting a small speedup, and a large speedup.

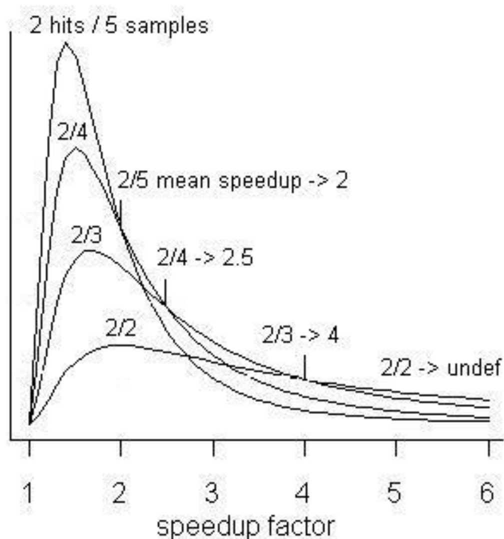

Figure 2. Distribution of Y, given *s/n*.

Speedup factor Y is $1/(1-X)$[Amdahl], and its distribution is $Y \sim BetaPrime(s+1, n-s+1) + 1$. It is shown in Figure 2.

Both of these probability distributions would show narrower peaks if n were larger, thus reducing the probability of false positives and small speedup. That's what summarizing profilers do. The price paid for that precision is less information about what the speedup actually *is*, thus less chance of finding it and getting *any* improvement. Such a failure to find a speedup is a "false negative". It is much more costly than a false positive, as shown below.

2.1. The Cost of False Negatives

It cannot be assumed a program contains only one possible speedup. Figure 3 shows a hypothetical program as an illustration. It originally takes 100 seconds. It contains six possible speedups, A through F, in a range of sizes. If all six speedups are effected, the overall speedup factor is 100/11.8 = 8.5. If one of them happens to be omitted, such as D, then the overall speedup factor is 100/(11.8+10.3) = 4.5. The ratio between speedups of 4.5 and 8.5 is the price paid for omitting D.

2.2. Magnification Effect

Consider speedup F. It is only 5%, so it is small, and would take on the order of 40 samples to see it twice. However, speedup A takes 30% of time, and the average number of samples needed to see it twice is 2/0.3 = 6.67 samples (negative binomial distribution). So 10 or

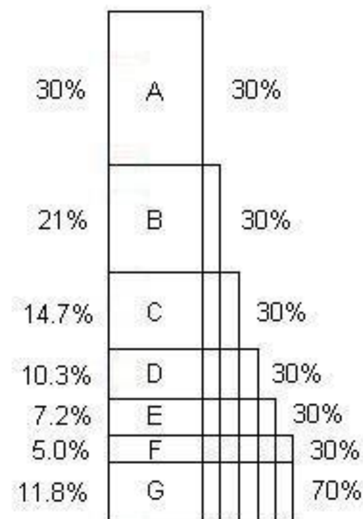

Figure 3: Hypothetical multiple speedups.

20 samples locate it with ease. Fixing it reduces execution time to 70 seconds. Now B is 30%, not 21%, because 21/70 = 0.3. (This reduction of the denominator can be called a *magnification effect*.) This means B is

easier to find on the second iteration of the technique. When all of A through E have been found and effected, F is no longer 5%, it is 30%, so it is easy to find.

3. The Example – Initial Time: 2700 usec

The purpose of this example is to show that there are speedups in real software that profilers would have trouble finding, it at all, that are found by the manual technique.

The example[Dunlavey] is a discrete simulation of a real program for sequencing tasks through an automated factory. It is about 800 lines of C++. It processes a series of work units called "jobs", and the primary measure of performance is the time per job, which initially was 2700 microseconds. Timing figures were gotten by running 10^4 and 10^5 jobs, and counting seconds, which is why the overall times have only two digits precision. This amount of time gave ample opportunity to take samples.

In the following stack samples, the content of every line is not shown, but attention is drawn to certain source code by including it and shading it.

3.1. First Iteration – Save 900 usec

Five stack samples were taken. Here are two of them:

```
main() Line 317
doit() Line 296
CJobReq::Handler() Line 103
  CBop* pBop = new CBop;
  pJob->bops.push_back(pBop);
std::vector<CBop *,std
    ::allocator<CBop *>>
    ::push_back() Line 823
std::vector<CBop *,std
    ::allocator<CBop *>>
    ::insert() Line 878

main() Line 317
doit() Line 296
COpReq::Handler() Line 131
  CTask* pTask = new CTask(i, (i % 2));
  pOp->tasks.push_back(pTask);
std::vector<CTask *,std
    ::allocator<CTask *>>
    ::push_back() Line 823
std::vector<CTask *,std
    ::allocator<CTask *>>
    ::insert() Line 878
```

Difficulty with profiling (DWP): If these five samples were typical of a large number, they would tell the summarizing profiler the *insert* function of the *allocator* class of the *std::vector* class was taking a lot of time. (That is of little use to the programmer, because the programmer cannot alter those functions). It shows that functions *vector<CBop*>::push_back*, and *vector<CTask*>::push_back* are taking time. They are not aggregated when summarized, due to templating. Similarly the source code lines 103 and 131 are not aggregated because they are different lines.

The human programmer can see that roughly 2/5 of time is spent in *push_back*, regardless of class. This suggests a speedup opportunity of about 40%. If the program runs for 60 seconds, it is spending around half of that time growing vectors for the purpose of pushing things on their backs. Surely there is a better way to do that.

There are different ways this can be remedied, all more or less effective. This paper is about how to find speedup opportunities, not about how to effect them. In this case, since it was done in Visual C++ using Microsoft Foundation Classes (MFC), the classes derived from *std::vector* were switched to *CTypedPtrArray*, hoping it would be more efficient. It was, saving 33%, giving an overall speedup factor of 1.5. This saves 900 microseconds, bringing the total down to 1800.

3.1. Second Iteration – Save 300 usec

On the second iteration, ten samples were taken. What stands out to the human programmer is that four of them are in *operator[]()*, the overloaded array indexing operator. (This was a debug build. It is true that if it had been a release build, this operator function would not have been used. However, the paper is about how to find speedups, not how to prevent them. A problem with release builds is that they are hard to debug, so let's just say that we effectively found and roughly quantified a problem that didn't really have to be there.)

DWP: The question is, would a profiler have found this problem? First of all, if it only summarized at the function level, it would have shown that the *operator[]()* function of the *CTypedPtrArray* class had about 40% inclusive time. That does not point to the lines from which the costly calls came. If the profiler summarizes at the level of lines of code, those lines do not aggregate much time because they are in at least four places scattered over the code.

```
main() line 318
doit() line 297
CMhAck::Handler() line 165
TcProcess() line 246
  COperation* pOp = oplist[i];
CTypedPtrArray<CPtrArray,COperation *>
  ::operator[]() line 1555

main() line 318
doit() line 297
COpAck::Handler() line 145
SchProcess() line 212
  pJob = joblist[i];
CTypedPtrArray<CPtrArray,CJob *>
  ::operator[]() line 1555

main() line 318
```

```
doit() line 297
CMhAck::Handler() line 165
TcProcess() line 249
  pTask = pOp->tasks[pOp->iCurTask];
CTypedPtrArray<CPtrArray,CTask *>
  ::operator[]() line 1555

main() line 318
doit() line 297
CTskAck::Handler() line 193
TcProcess() line 259
COperation::
  `scalar deleting destructor'()
COperation::~COperation() line 57
  ~COperation(){
    for (int i = tasks.GetSize();
        --i>=0;)
    {
      CTask* p = tasks[i];
      tasks[i] = NULL;
      delete p;
    }
  }
CTypedPtrArray<CPtrArray,CTask *>
  ::operator[]() line 1555
```

At any rate, the human programmer can easily see that 40% of the time is spent in calling the vector indexing function, and that a simple fix is to switch to direct indexing. This was done, saving 300 microseconds or 17%, bringing the total down to 1500, with a speedup factor of 1.2. It was not a problem that the percent was wrong, because the speedup was found and effected.

3.1. Third and Fourth Iterations – Save 200 and 860 usec

Ten samples were taken. Of them, six were noted.

Two of them were doing *new*:

```
main() line 367
doit() line 346
COpReq::Handler() line 139
  COperation* pOp =
    new COperation(iNextOp++, jobid);
operator new() line 65
operator new() line 373

main() line 367
doit() line 346
CMhAck::Handler() line 188
TcProcess() line 320
  else if (pTask->type == 1){
    transactions.Add(
      new CTskReq(pOp->id));
  }
operator new() line 65
operator new() line 373
```

Two of them were doing *Add*:

```
main() line 367
doit() line 346
COpReq::Handler() line 146
TcProcess() line 319
    transactions.Add(
      new CMhReq(pOp->id));
CTypedPtrArray<CPtrArray,CTran *>
  ::Add() line 1539
CPtrArray::Add() line 172
CPtrArray::SetAtGrow() line 183

main() line 367
doit() line 346
CTskReq::Handler() line 233
    transactions.Add(
      new CTskAck(tskid));
CTypedPtrArray<CPtrArray,CTran *>
  ::Add() line 1539
CPtrArray::Add() line 172
CPtrArray::SetAtGrow() line 183
```

Two of them were running destructors:

```
main() line 367
doit() line 346
CTskAck::Handler() line 225
TcProcess() line 304
  // delete operation from oplist
  oplist.RemoveAt(i); --i;
  delete pOp;
COperation::
  `scalar deleting destructor'()
COperation::~COperation() line 69
CTypedPtrArray<CPtrArray,CTask *>
  ::~CTypedPtrArray<CptrArray
      ,CTask *>()
CPtrArray::~CPtrArray() line 47
operator delete() line 351

main() line 367
doit() line 346
CTskAck::Handler() line 225
TcProcess() line 304
    oplist.RemoveAt(i); --i;
    delete pOp;
COperation
  ::`scalar deleting destructor'()
COperation::~COperation() line 67
operator delete() line 351
```

Two changes were done. The first change was, since objects were always appended to vectors, and removed from the front, the vectors could be replaced by linked lists. This was done, saving 200 microseconds or 13%, bringing the total down to 1300, for a speedup factor of 1.15.

There is a key point here. This change makes irrelevant the changes done in the first two iterations. It only works if the programmer is persistent about changing things, regardless of prior investment. If she is "attached" to the prior code, the whole process stalls.

The second change was to recycle used objects in free lists, so as to reduce memory allocation and deallocation. This saved 860 microseconds or 66%, bringing the total down to 440, for a speedup factor of 2.95. As can be said of every speedup after the first, its speedup factor depends on prior speedups having been effected.

DWP: Profilers that summarize by line of code would not be able to aggregate the similarities between the samples, because they occur in multiple locations and call multiple functions.

3.1. Fifth Iteration – Save 270 usec

Five samples were taken. Of them, three were noted.

```
main() line 367
doit() line 530
CTran::Handler() line 277
CTran::HandleOpAck() line 366
SchProcess() line 418
  NTH(pTask, pOp->tasks, pOp->iCurTask);

main() line 367
doit() line 530
CTran::Handler() line 279
CTran::HandleMhAck() line 387
TcProcess() line 468
  NTH(pTask, pOp->tasks, pOp->iCurTask);

main() line 367
doit() line 530
CTran::Handler() line 276
CTran::HandleOpReq() line 356
TcProcess() line 468
  NTH(pTask, pOp->tasks, pOp->iCurTask);
```

These three samples were all in the NTH macro, a macro that had been added as part of the conversion to linked lists. It performs the indexing operation on a linked list.

DWP: A profiler that summarizes at the level of lines of code would have aggregated two of these, but not the third. The similarity is obvious to a human who looks at the code.

The solution to this is, since the indexes into the lists were simply stepped along sequentially, to use pointers to point directly into the lists, rather than integer indexes. This saved 270 microseconds or 62%, bringing the total down to 170, for a speedup factor of 2.6.

3.1. Sixth Iteration – Save 166 usec

Four samples were taken. All four looked like the following, meaning they accounted for a fraction of time close to 100%:

```
main() line 367
doit() line 346
CJobAck::Handler() line 126
  cout<<"Ack Job "<<jobid<<endl;
std::basic_ostream<char,std
  ::char_traits<char>>
  ::operator<<() line 115
std::num_put<char,std
  ::ostreambuf_iterator<char,std
  ::char_traits<char>>>
  ::put() line 444

... five more layers ...

std::basic_filebuf<char,std
  ::char_traits<char>>
  ::overflow() line 108
std::_Fputc() line 42
```

DWP: This is a typical stack sample during I/O. Profilers that sample on CPU time instead of wall-clock time suspend sampling during I/O, so are blind to it. Only if they sample on wall-clock time, and summarize inclusive time at the level of lines of code, would they highlight the responsible line.

The human examining the samples is led directly to that line. Furthermore, the stack lines above that line give the reason why the I/O is being done, so she can decide if it is truly necessary.

Whether the I/O was necessary depends on the situation. (I have certainly seen cases where it was not.) Regardless, the point is that *it was found, and found to be significant*, so the choice could be made. In this case, it was decided that the I/O was not necessary, so it was commented out. This removed 166 microseconds or 97.8% of the time, bringing the total down to 3.7, for a speedup of 45.9.

4. Conclusions

The final execution time, of 3.7 microseconds per job, is 730 times faster than the original.

It is fair to say that the initial performance would have been much better if a release build had been used, and if the I/O had been commented out at the beginning. In that case the initial time would have been 2700 – 900 – 300 – 166 = 1334 microseconds, basically twice as fast as the original (in which case the speedup would have been only 365 times). We are still left with the task of finding the remaining speedups, the ones taking 200, 860, and 270 microseconds, bringing the time down to 3.7.

References

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

Ammons, G., Ball, T, Larus, J., Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN PLDI,* June 1997

Bentley, J., Writing Efficient Code, Carnegie Mellon University, 1981.

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

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

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

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

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

Liang, S, Visnawathan, D, Comprehensive Profiling Support in the Java Virtual Machine, USENIX COOTS '99, May 1999.

Zoom, Rotate Right, http://www.rotateright.com.