Differential Profiling

Paul E. McKenney
Sequent Computer Systems, Inc.
(now with IBM)
pmckenne@us.ibm.com

Abstract

Performance can be a critical aspect of software quality; in some systems, poor performance can cause financial loss, physical damage, or even death. In such cases, it is imperative to identify system performance problems before deployment, preferably well before implementation.

Unfortunately, the size of most software systems grossly exceeds the capacity of current performance-modelling techniques. Hence, there is a need for techniques to quickly identify the portions of the system that are performance-critical. These portions are often small enough to be modelled directly.

This paper describes one such technique, differential profiling. Differential profiling combines two or more conventional profiles of a given program run in different situations or conditions. The technique mathematically combines corresponding buckets of the conventional profiles, then sorts the resulting list by these combined values. Different combining functions are suitable for different situations.

This combining of conventional profiles frequently yields much greater insight than could be obtained from either of the conventional profiles. Hence, differential profiling helps to locate difficult-to-find performance bottlenecks, such as those that are distributed widely throughout a large program or system, perhaps by being concealed within macros or inlined functions.

This paper also describes how this technique may be used to pinpoint certain types of performance bottlenecks in large programs running on large-scale shared-memory multiprocessors. In this environment, the critical bottleneck might consume only a small fraction of the total CPU time, since typical critical sections can consume at most one CPU's worth of computation. This sort of bottleneck, particularly when widely distributed throughout the program under consideration, is often invisible to traditional profiling techniques.

Keywords: parallel performance analysis profiling

Introduction

Performance tuning of large-scale systems is still somewhat of an art [1]. One reason for this is that the size of current software systems far exceeds the capacity of current analytic techniques. For example, a system with 50,000 critical sections running on a 32-CPU hardware platform has more than

$$\binom{50,032}{32} > 10^{114} \tag{1}$$

states, since any or all of the CPUs might be within or waiting for any of the critical sections, but CPUs are indistinguishable, as are critical sections containing no CPUs. Clever state-collapsing techniques might result in only 100 effective critical sections, which in turn results in "only"

$$\binom{132}{32} > 10^{30} \tag{2}$$

states. And even this number assumes that only the lock-contention behavior of the program is relevant. If other aspects are important, the state space becomes even larger.

Any feasible performance tuning effort must concentrate on a very small subset of these states. Differential profiling may be used to help identify this subset, thereby forming a bridge between large real-world systems and powerful, highly-focused analytic techniques.

Classic profiling has been widely used for decades to help locate performance bottlenecks. It is nonetheless useful to quickly review this almost-reflexively-used technique before introducing differential profiling.

A given type of profiling drastically filters a program's execution history to provide the needed information within a finite storage budget–and the filtering must be drastic indeed if the program visits even a tiny fraction of its possible states. Most profiles form a histogram of program-counter values, sampled ei-

ther deterministically or randomly[2].[1] The program-counter information may be augmented with function-call counts in order to get gprof-style profiling [5]. Machine-specific knowledge may be incorporated into a toolset in order to help the user determine what types of overhead are affecting a given segment of code [6]. Reiser and Skudlarek show how to use a particular variant of differential profiling to find localized algorithmic bottlenecks [7]. In contrast, this paper presents generalized differential profiling and shows how to use this technique to locate bottlenecks that are diffused throughout a program or system, such as those caused by data or lock contention in parallel programs.

This paper describes some methods of extracting more information from whatever data is retained, particularly for parallel programs and systems. For concreteness, this paper takes most of its examples from the conventional sampled-program-counter variety of profiles.

The following sections give an overview of differential profiling, demonstrate use of multiple profiles to pinpoint performance problems on several "toy" problems, and describe experiences using differential profiling on large programs.

Differential Profiling

The basic idea behind differential profiling is to collect measurements at multiple load levels (bucketed into sets of interest, e.g., by function or by machine instruction) and compare corresponding buckets of these data sets. The most common comparison methods are ratio, weighted differences, and projected saturation.[2]

Ratio Differential Profiling

Use the following procedure to produce a ratio differential profile:

1. Collect a pair of profiles from a workload run under different conditions.[3]
2. Compute the ratio of the measurements from corresponding buckets of the two profiles. Place the measurements taken from the more stressful[4] of the two runs in the numerators of the ratios.
3. Sort the result in descending order of ratio values.

The buckets that sort to the beginning of the list are likely to be the ones most affected by whatever problem caused the increased stress, be it cache thrashing, data contention, or simply poor choice of algorithm. Therefore, analyzing the system in the order that its parts appear in the ratio differential profile is almost always a good way to quickly locate problems.

Locating the source code and data structures corresponding to those buckets can help pinpoint those data structures and algorithms that can benefit most from optimization effort, even in cases where their use is diffused throughout a system, for example, by macros or inlined functions.

The parts of a system that have similar scaling problems will often sort to the same location in a ratio differential profile. For example, the parts of the system with $O(n^3)$ complexity would sort before the parts with $O(n^2)$ complexity, where n is a measure of the condition being varied, such as the number of data items processed by the system. For this reason, the ratio differential profile can be thought of as the software equivalent of the spectrascope. It takes profile data that would otherwise be white noise and reveals the inner structure of the program, just as the spectrascope allows astronomers to determine the composition of stars from the white light emanating from them.

Weighted Differential Profiling

A ratio differential profile is the most versatile, and is therefore usually the first choice in a new situation. However, a weighted differential profile is easier to use when you can: (1) quantify the amount of work

[1] In general, a profile is a histogram of some measurement bucketed by regions of interest. A traditional profile uses number of profiling-interrupt "hits" as the measurement, and functions as the buckets. Entirely different measurements are possible and often useful: disk activity, memory consumption, cache misses, and other hardware operations [3, 4] might be measured instead of the traditional profiling hits. In fact, a single profile might well contain several different types of measurements.

The measurements might be bucketed by instruction, module, data structure, or source-code line rather than by function.

[2] There are any number of additional variations on the differential-profiling scheme, but these three appear to be the most generally useful. In some cases, non-parametric statistics [8] can be helpful.

[3] Any number of conditions might be varied, for example, number of CPUs, number of disks, amount of memory, number of network interfaces, number of data items, number of users, transaction rate, network packet rate, and so on. Choose the condition to vary based on the issue at hand. For example, to determine how well the program scales on a multiprocessor system, vary the number of CPUs.

[4] The concept of "more stressful" depends both on the workload and the part of the system under test. For example, increasing the number of network interface cards places more stress on the interface-selection code, but less stress on protocol queueing and retransmission algorithms.

done in each run,[5] (2) show that the profile measurements relate directly and linearly to this amount of work done, and (3) run full-size workloads.

The procedure for a weighted differential profile is identical to that for a ratio differential profile except that the quantity:

$$d = w_1 m_2 - w_2 m_1 \quad (3)$$

is used for each bucket in place of the ratio, where m_i and w_i are the measurement and the amount of work done, respectively, in the current bucket of the ith workload. Workload 1 is the least stressed, and workload 2 is the most stressed.

The weighted differential profile's ease of use is due to the fact that it automatically filters out measurements corresponding to code that scales extremely poorly, but that consumes so few resources that the poor scaling is irrelevant. This filtering is quite desireable when you wish to optimize a program and can run it under full load.

In contrast, ratio differential profiles would highlight these poorly-scaling code fragments. Therefore, it is often necessary to filter out these "false alarms", perhaps by rejecting any bucket whose measurements both fall below some cutoff point. However, this highlighting is exactly what you want if you are trying to eliminate scaling problems in cases where you cannot run a full workload. In such cases, you are using differential profiling to extrapolate to a higher level of stress. The well-known hazards of extrapolation are discussed at the end of the next section.

Projected-Saturation Differential Profiling

Ratio and weighted differential profiling work best in cases where the measurements could increase indefinitely. For example, as you add CPUs, system-wide CPU consumption can increase, limited only by the number of CPUs you add.

In constrast, measurements such as per-disk I/O rate, lock hold time, and network packet rate are limited by the capacity of the entity being measured. In these cases, attention must focus on the measurement that is expected to reach saturation with the smallest increase in load. This might not be the same measurement whose ratio or difference is increasing most quickly. For example, Table 1 shows hypothetical utilizations measurements of three resources (CPU, memory, and disk) taken at two different loads, along with their ratios and differences. The "L=1" and "L=2" columns show saturation of the resources at one and two units of load, respectively. The "R" column shows the ratio of the saturations of the resources at the two loads. The "D" column shows the weighted difference from Equation 3, where w_1 is one unit of load, w_2 is two units of load, and m_1 and m_2 are the two measured saturations. The "LSL", or linear saturation load, column shows the number of units of load that would cause the corresponding load to be 100% saturated, assuming that saturation is a linear function of load. Note that the disk resource, which has

	L=1	L=2	R	D	LSL
CPU	5%	10%	2.00	5	20
Memory	20%	30%	1.50	10	9
Disk	98%	99%	1.01	1	3

Table 1: Ratio, Difference, and Saturation Profiling

the smallest ratio and difference, will reach saturation first, so in this (contrived) example, ratio and difference differential profiling would be misleading.

Therefore, a projected-saturation differential profile should be used in cases where the measurements are subject to saturation.

The procedure for a linear projected-saturation differential profile is identical to that for a ratio differential profile with two exceptions. First, the quantity:

$$s = \frac{(m_s - m_2)(l_2 - l_1)}{m_2 - m_1} + l_2 \quad (4)$$

is used for each bucket in place of the ratio, where m_i and l_i are the measurement and the offered load, respectively, in the current bucket of the ith workload. The m_s is the saturation measurement. Second, the result must be sorted in ascending rather than descending order.

There are many projection methods in addition to linear. In cases where the entities being measured have different asymtotic complexities (e.g., $O(n)$ vs. $O(n^2)$), a power-law projection may be helpful:

$$s = \frac{log(m_s/m^2) log(l_2/l_1)}{log(m_2/m_1)} + log(l_2) \quad (5)$$

Other projection methods may require measurements at more than two loads.

[5] The definition of "amount of work done" is application specific. "Work done" might be the number of searches performed for a data-structure-traversal algorithm, transactions per second for a database system, packets per second for a network protocol implementation, I/Os per second for a disk driver, number of iterations for a scientific workload, number of users supported with a given response time for an operating system, and so on. A proper definition of "amount of work done" is crucial to any performance-analysis effort. If you don't know what the system is supposed to be doing, you will likely have some difficulty optimizing its performance.

Projected-saturation differential profiling is in effect using extrapolation to estimate resource consumption at higher loads than were measured. In practice, measurement errors, utilization of multiple resources, and changes in behavior can often render the always-dangerous process of extrapolation completely foolhardy.

There are nevertheless situations where the risks involved in this sort of extrapolation can be tolerated. For example, careful performance extrapolation might be used to help argue for the equipment and labor expenditures required to test at a higher load than could otherwise be justified. In addition, careful performance extrapolation can help locate performance bugs on smaller, cheaper machines than would otherwise be required. Extrapolation does not completely remove the need to test on expensive full-sized machines, but it can reduce the amount of test time required, given that fewer performance bugs remain to be found, fixed, and retested. There are many techniques and procedures from the field of capacity planning [9] that can reduce the risks associated with performance extrapolation.

However, a later section shows that a class of algorithms commonly used in parallel programs can make simple CPU-time extrapolation extremely dangerous. That section also shows how to more safely extrapolate the performance of this class of algorithms.

Example Profiling Exercises

This section demonstrates use of differential profiling to pinpoint poorly-scaling algorithms, cache thrashing, and lock contention.

Pinpointing Excessive CPU Usage

The first example demonstrates the use of differential profiling to find a poorly-scaling algorithm in a toy program. Actual measurements for loads of 1, 2, 3, and 4 are plotted in Figure 1. The consumption appears to be fairly linear with the load, where "load" is defined as the value of a parameter n.[6] However, an actual run at a load of 10 consumes over *350* CPU seconds.

Ratio differential profiling, displayed in Table 2, may be used to quickly locate this scaling problem *without* actually running the program at a load greater than 4. The ratio focuses our attention on the problem area–the `log` and `poly1` functions are exhibiting near-linear increases in execution time, but the `poly2` function is clearly growing very rapidly.

[6] This type of definition of "load" is often found in scientific and engineering simulations, where a larger value of n might indicate a finer mesh size, with correspondingly larger data sets to be operated on.

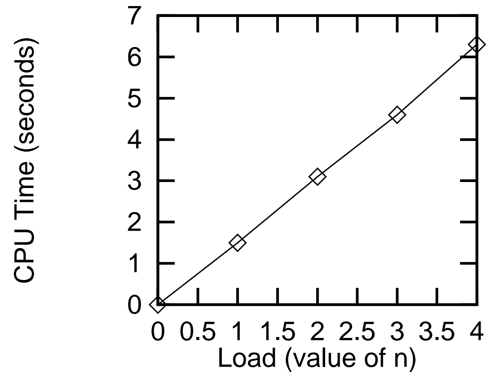

Figure 1: CPU Time for Example 1

Function	Load=3	Load=4	Ratio
poly2	1	12	12.00
log	409	554	1.35
poly1	60	70	1.16

Table 2: Differential Profile

The source code to the `poly1` and `poly2` functions are displayed in Figures 2 and 3, respectively. The `poly2` function has time complexity $O(n^9)$, which fully accounts for poor scaling–at a load of 10, the `poly2` function accounts for more than 95% of the CPU consumption.

Note that this technique of sorting on ratios of execution times can help focus on the most critical areas even if there are tens of thousands of functions in an existing program with no documentation and no gurus.

```
for (i = 0; i < n; i++)
    for (j = 0; j < 100000; j++) {
        x = log(x);
        if (x <= 1.)
            x = x + 5.;
    }
```

Figure 2: `poly1` – Linear Time Complexity

A potential complication is that a large number of functions may be executed so infrequently that they only accumulate a few counts on any given run. These functions can have very large ratios merely due to statistical fluctuations. The resulting false alarms may

```
for (i1 = 0; i1 < n; i1++)
    ...
        for (i9 = 0; i9 < n; i9++)
         sum++;
return (sum);
```

Figure 3: poly2 – $O(x^9)$ Time Complexity

be eliminated by rejecting functions that do not have at least one profiling count that is well above the level of measurement error.

Pinpointing Cache Thrashing

Cache thrashing occurs when a parallel algorithm causes CPUs to frequently reference data items that are often modified by other CPUs. Each such data item will "thrash" back and forth across the global bus or interconnect connecting the CPUs' local caches. Since global processing is very costly in comparison to local processing and is expected to become even more expensive as technology advances [10, 11, 12], cache thrashing causes dramatic decreases in performance.

Thrashing is demonstrated by the code fragment shown in Figure 4. This code fragment searches for a pair of control blocks in separate lists, incrementing a counter in each of the control blocks that it finds. If one or the other of the control blocks could not be found, the corresponding pointer will be NULL. Similar code fragments are used to keep separate local (per-CPU) and global state. Such separate local and global state can be used to detect and optimize cases where the corresponding entity happens to be used only by a single CPU.

In this example, "load" is the number of CPUs, and the measure of work is the number of searches performed. The number of searches per CPU is held constant, so that the amount of work done is proportional to the number of CPUs.

If this code fragment scaled perfectly, search time would not depend on the number of CPUs. However, the two-CPU run of this code fragment consumed about 19% more CPU per process than did the single-CPU run. Although this is not disasterously greater CPU consumption, this is only a two-CPU run. Disaster very likely awaits a four- or eight-CPU run.

A standard per-source-line profile of the two-CPU run is shown in Table 3. Four of the top five lines are from the second `for`-loop from Figure 4, hinting that the data structure searched by this loop be restructured.

```
 7  struct ctrlblk {
 8      struct ctrlblk *next;
 9      int     id;
10      int     ctr;
11      char    pad[20];
12  };

...

61  /* Search for control blocks. */
62
63  for (p = tp; p != NULL; p = p->next) {
64      if (p->id == key) {
65          p->ctr++;
66          break;
67      }
68  }
69
70  for (q = tpp; q != NULL; q = q->next) {
71      if (q->id == key) {
72          q->ctr++;
73          break;
74      }
75  }
```

Figure 4: Cache-Thrashing Code Fragment

Line	Profiling Hits
73	556
64	519
75	457
70	417
71	206
68	162
63	142
66	108
65	21
72	18

Table 3: Two-CPU Conventional Profile

However, a weighted differential profile, shown in Table 4, tells a different story. The last column of this table is computed using Equation 3, where w_1 is 1 CPU,[7] w_2 is 2 CPUs, m_1 is the 1-CPU profile measurement, and m_2 is the 2-CPU profile measurement. Line 64 is the only line whose overhead increased dramatically from the single-CPU to the dual-CPU run. Line 64 is the first line to reference the next element in the linked list in the first loop, which hints that there may be a cache-thrashing problem.

Line	# CPUs 1	2	Weighted Difference
64	61	519	397
75	198	457	61
68	74	162	14
71	96	206	14
72	4	18	10
76	13	25	-1
73	279	556	-2
65	16	21	-11
70	214	417	-11
66	68	108	-28

Table 4: Differential Profile

The problem is that the increment on line 65 invalidates the cache line, removing it from all other CPU's caches. When one of these other CPUs next tries to reference this line, it must fetch a new copy. Since the second loop (lines 60-74) searches a list that is private to each CPU, this loop does not suffer from this cache-thrashing effect.

The traditional way to prevent cache thrashing in this case is to place the `ctr` field into its own cache line as shown in Figure 5. This placement prevents modifications to the `ctr` field from invalidating the `id` and `next` fields that are used in the search. The

```
 7  struct ctrlblk {
 8      struct ctrlblk *next;
 9      int     id;
10      char    pad1[24];
11      int     ctr;
12      char    pad2[28];
13  };
```

Figure 5: Fix to Cache-Thrashing Code Fragment

[7] Since the amount of work done is proportional to the number of CPUs, the numbers of CPUs may be used directly as the weights.

modified program runs slightly slower on a single CPU (11.6 CPU seconds compared to 11.5 for the original version), but runs much faster for greater numbers of CPUs, as shown in Figure 6.

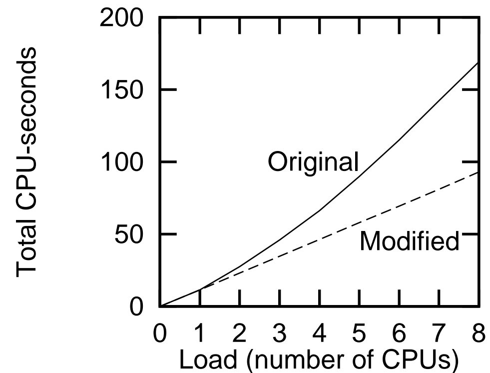

Figure 6: Original vs. Modified Implementation

In this example, ratio differential profiling correctly identified the cause of poor scaling. In contrast, conventional profiling gave misleading information.

Pinpointing Contention

This example demonstrates the use of differential profiling to locate lock contention, again in a toy program. Figure 7 shows an example program that is subject to contention. The `s_lock_p` function is an instrumented version of the `s_lock` function that acquires a spinlock, and the `s_unlock_p` function is likewise an instrumented version of the `s_unlock` function that releases a spinlock [13].

```
s_lock_p(&spinlock);
p = idp;
for (; p->id_next != NULL; p = p->id_next) {
    p->id_ctr++;
}
s_unlock_p(&spinlock);
```

Figure 7: `contend` – Lock Contention

In this example, "load" is defined to be the number of elements in the linked list pointed to by `idp`, in thousands.

The fraction of time spent spinning is shown as a function of load in Figure 8. This number increases sharply and suddenly when the load reaches 1.4. Thus, the fraction of CPU time spent spinning is not a good

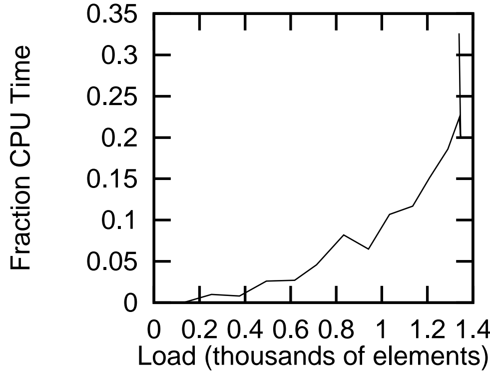

Figure 8: Fraction CPU Time Spinning

predictor of the maximum load that can be sustained.[8]

To see why, consider that short, frequently-occuring critical sections are often guarded by spinlocks. Such critical sections running on a shared-memory parallel processor will typically spin for a time period approximated by:

$$f(x) = \frac{rsx}{1 - sx} \tag{6}$$

Here, $f(x)$ is the spin time not including the time required to execute the critical section and surrounding code, r is the average time required to execute the critical section and s is a constant that converts a load x into a utilization such that when the utilization is 1.0 the critical section is saturated[9]. This saturation will occur when the critical section consumes one full CPU. Thus, on a machine with a large number of CPUs, the CPU overhead of the worst critical section excluding spinning might rank quite low, and therefore might not stand out on a conventional profile.

One might hope that the spinning overhead *would* stand out. To extrapolate this overhead, we take a measurement of the time spent spinning at some load b. The presence of measurement errors means that a zero measurement must be interpreted to mean that the spin time $f(x)$ at load b is less than the measurement error (call it ϵ). It is often sufficient just to show that the spin time $f(x)$ at the higher load c will be within some budget M.

Substituting Equation 6 into the measurement-error and budget constraints yields the following solutions for r and s whenever $M/\epsilon > c/b$:[10]

$$r = M\epsilon\frac{c - b}{Mb - \epsilon c}, s = \frac{Mb - \epsilon c}{Mbc - \epsilon bc} \tag{7}$$

Hence, it is possible for an algorithm to not spin at all below load b but still exceed an arbitrarily large budget at load c, regardless of how closely-spaced b and c are. Therefore, simple profile-based extrapolation just does not work for predicting critical-section spin times. The author has personally witnessed several algorithms whose spin times "blow up" in response to small increases in load.

However, the behavior of parallel algorithms *may* often be reasonably safely extrapolated by estimating the values of r and s directly from the code itself. These estimates may be accomplished either by directly measuring the quantities, perhaps by using a high-accuracy clock, or by computing them from profiling data.

Usually, the value of interest is sx, the CPU utilization of the critical section. When this value approaches one, the corresponding critical section becomes a bottleneck. The value of sx for the example is shown in Figure 9. This fraction clearly cannot rise

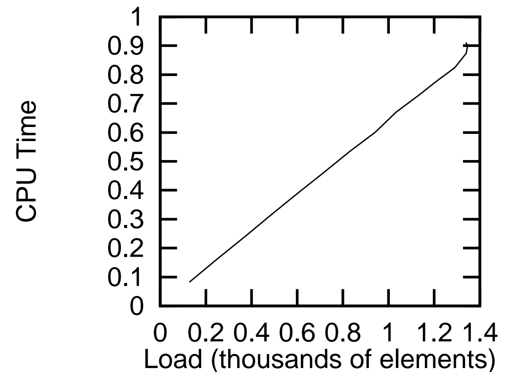

Figure 9: Fraction of One CPU Consumed by Critical Section

above 1.0 for any given critical section. In fact, in this case, it does not even rise much above 0.9. This is due partially to measurement error but mostly because variations in execution rate can cause the CPUs to all be executing outside of the critical section. This time spent with no CPUs in the critical section can never be made up.

Figure 10 shows that the spin time is fit very closely by a function of the form of Equation 6 with r equal to 0.05 and s equal to 0.73.

[8] However, spin time *can* be helpful in diagnosing an existing performance problem.

[9] This equation assumes that the critical section is entered at a exponentially-distributed rate and executes at an exponentially-distributed rate. This equation is a reasonable approximation for other distributions, as can be seen from graphs of this and related functions [14]. Queuing theorists will note that r is $1/\mu$, sx is the utilization λ/μ (sometimes denoted by ρ), and that $f(x)$ itself is the in-queue waiting time W_q.

[10] This condition will normally hold. The budget M *should* be very large compared to the measurement error ϵ – otherwise, you should be using a better measurement methodology.

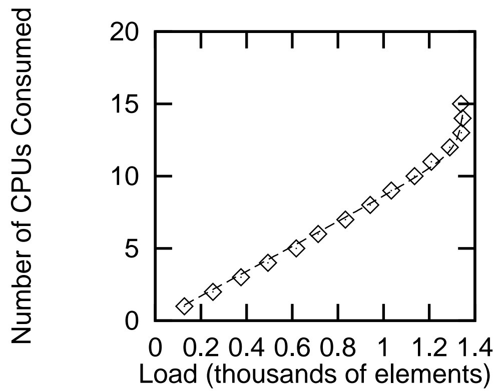

Figure 10: CPUs vs. Load Fit

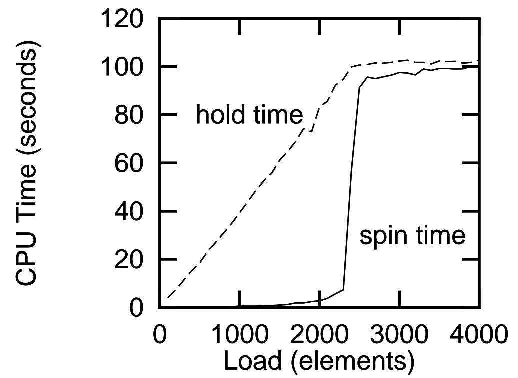

Figure 12: Spin/Hold Times For Periodic List Scan

As noted earlier, spin times can suddenly "blow up" in response to small changes in offered load. This effect is demonstrated by the code fragment in Figure 11. This code fragment is invoked every clock tick, each time on a different CPU. The offered load is simply the number of elements in the linked list. The larger the number of elements, the more CPU time will be consumed in the critical section. Figure 12 shows that the spin time increases sharply and suddenly, with no warning. However, the hold time gives a good picture of scaling limits.

```
s_lock_p(&lock1);
p = idp;
while (p->id_next != NULL) {
        p->id_ctr++;
        p = p->id_next;
}
s_unlock_p(&lock1);
```

Figure 11: Periodic List Scan

This example demonstrates the pitfalls of naively extrapolating the CPU consumption of code that is subject to contention. This section has demonstrated an alternative, more robust method:

1. Instrument locking primitives to measure the amount of time that each lock is held.
2. If necessary, also account for the time required for the lock to move from one CPU to the next.
3. Divide each such measurement by the duration of the run to obtain the fraction of time that each lock was held.
4. Apply projected-saturation differential profiling to the resulting fractions.

This method allows you to see which locks are nearest to becoming saturated and thus becoming bottlenecks, even if the corresponding critical sections are scattered widely throughout the code.

Differential-Profiling Cautions

Differential profiling located these problems readily. However, some caution is required when using differential profiling on the large-scale systems found in the "real world". No technique, not even differential profiling, will ever be an acceptable substitute for careful thought, planning, and analysis.

First, it is not uncommon for large-scale systems to use different algorithms at different load levels. For example, it is quite common for parallel programs to have special-case code for single-CPU operation. In these cases, it is necessary to compare two-CPU and three-CPU runs rather than one-CPU and two-CPU runs. In general, you must take care either to compare identical systems or to allow for any differences that might appear.

Second, obtaining accurate and meaningful measurements often requires much care and creativity. This is especially true in distributed applications, where the effects of communications overheads can be difficult to measure properly. For example, in a distributed lock manager, the time taken to communicate the availability of a lock to the next process wishing to acquire it should be counted as time that the lock is unavailable. Otherwise, you will get an overly optimistic estimate of the system's scalability. This effect is not limited to distributed systems; the relatively long latencies found in large-scale SMP systems can also distort lock-hold-time measurements.

Obtaining accurate and meaningful measurements in monolithic systems can be just as challenging. For example, the overhead of taking the measurements might completely change the behavior of the system. To guard against this, it is wise to perform an unprofiled run so that you can determine whether the act of measurement is changing the system's behavior.

Third, especially when using a ratio differential profile, statistical variations can cause false alarms. A given code fragment might, just by chance, collect one profiling hit on the first run and three hits on the second run. This coincidence could sort this irrelevant code fragment to the top of the list. It is therefore often necessary to filter a ratio differential profile to eliminate buckets that have too few hits to be statistically significant, or, if possible, to use a weighted differential profile instead.

If this filtering eliminates data that is known to be of interest, collect more data either by running the workload for a longer time period or by summing the profiles of several runs. If it is not possible to collect a large enough sample, it may be possible to get a valid comparison using comparison methods other than ratio or weighted difference. For example, nonparametric statistics [8] can sometimes be helpful.

Fourth, cache-capacity effects can confound the measurements. For example, if a two-CPU run uses more data than does a one-CPU run, the differential profile may locate cache misses due to the larger dataset no longer fitting into the CPU caches. Such cache misses may or may not be of interest; if they are not, then it is necessary to select the dataset sizes carefully in order to avoid this effect. This problem is not confined to hardware memory caches–the same problem can occur when profiling systems involving software caches for disk I/O or search structures.

Finally, hardware effects such as speculative execution and interrupt masking (for interrupt-based profilers) can produce misleading results. Hopefully, CPU designers will more carefully consider the needs of performance analysts in future designs.

Case Studies

The following three sections each describe a situation where differential profiling allowed developers to quickly locate a scalability performance problem. The first section describes a software bottleneck exposed by FDDI, the second section describes a process-creation bottleneck, and the third section describes an unusual cache-thrashing bottleneck.

FDDI Bottleneck

A pair of FDDI boards achieved only about 1.6 times the throughput of a single FDDI board, compared to the expected factor of 2.0. A conventional profiling report produced no useful insights; there was no obvious single point in the code responsible for the increased overhead.

Combining a profile of a system running a single FDDI board with that of a system running two boards pinpointed the problem. The time required to perform a particular hardware operation quadrupled as the offered load increased by a factor of 1.6. However, the operation itself accounted for less than 1% of the total CPU time consumed on the two-board benchmark, and is thus not noticeable on either of the profile reports when considered separately.[11]

However, a ratio differential profile put the offending operation right at the top of a list of well over a thousand functions.

A later version of the hardware eliminated the need for the offending operation.

Process-Creation Bottleneck

A benchmark showed disappointing process-creation rates: beyond a certain point, adding more CPUs and memory did not result in increased ability to create and destroy processes via the UNIX fork() system call.

A conventional profile simply pointed out the "usual suspects" responsible for much of the overhead of creating processes.

Combining a profile at low process-creation rate and at high process-creation rate demonstrated that some of the memory-allocation algorithms were subject to cache-thrashing at high load. Almost all of the increase in CPU overhead responsible for the poor scaling could be attributed to the five instructions used to copy the old process image while creating the new process image, and the combined profile put these instructions at the top of the list despite their being widely distributed throughout the code.

This information enabled developers to design new algorithms with better cache locality.

I/O Bottleneck

A pair of quad-SCSI interface boards was unable to perform more I/Os per second than was a single board. Since all previous versions of the operating system *did*

[11] The dramatic reduction in throughput was due to an interaction between the hardware operation and DMA.

scale I/Os per second with number of interfaces, suspicion fell on software.

Conventional profiling failed to find any significant CPU consumption. However, measurements of the system bus showed that the new software saturated the bus. Unfortunately, the hardware provides no reasonable way to assign bus utilization to software modules.

A differential profile comparing a single- and dual-interface benchmark run showed that a single instruction in the idle loop[12] consumed far more CPU time in the dual-interface run than in the single-interface run. However, it did not consume enough CPU time to stand out in a conventional profile.

Further analysis showed that this instruction was looking for newly-runnable processes as part of a new process-scheduling algorithm. However, this algorithm was thrashing the caches, resulting in excessive bus utilization, which in turn limited the bus bandwidth available for disk I/O.

When the process-scheduling algorithm was modified to avoid this cache-thrashing behavior, disk I/O again scaled with increasing numbers of quad-SCSI interface boards.

Conclusions

This paper showed how differential profiling can provide invaluable insights into the behavior of a program, by focusing attention on a small portion of the program so that powerful analysis techniques may be brought to bear. This technique may be used to locate conditions such as inefficient algorithms, spinlock contention, and cache-thrashing in large programs for which gurus, and perhaps even source code and documentation, are not available. In addition, this technique can make use of any type of profiling data. No new measurement tools are required.

The technique was demonstrated on several small example programs. Experiences successfully using it on real-world programs were discussed.

In the future, we expect to adapt this technique to more types of profiling data such as cache misses, and to more difficult situations, such as where a given function's overhead must be charged to its caller.

[12] This was the only time I ever conducted a performance analysis of the idle loop, something I had never expected to do. However, one of the anonymous referees is aware of similar occurrences in least two other systems.

Acknowledgments

I owe thanks to Noelan Olson, Brent Kingsbury, and Tony Petrossian for involving me in the situations that gave birth to the idea of differential profiling. Ken Dove implemented the fine-grained profiling system that made fine-grained differential profiling possible. Numerous conversations with Jack Slingwine helped me better understand the different approaches to profiling and how they are related to the ideal "complete" profile. I am grateful to Joseph Skudlarek, Jon Inouye, and Phil Krueger for their careful review of early drafts of this paper, to James Bash for helping to render it human-readable, and to Kirk Bailey for naming it.

Finally, I am indebted to Dale Goebel for his support of my efforts in this area.

References

[1] Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[2] Steven McCanne and Chris Torek. A randomized sampling clock for CPU utilization estimation and code profiling. In *USENIX Conference Proceedings*, Berkeley CA, February 1993.

[3] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, New York, NY, October 1997.

[4] Jeffrey Dean, James E. Hicks, Carl A. Waldspurger, William E. Weihl, and George Chrysos. Profileme: Hardware support for instruction-level profiling on out-of-order processors. In *Proceedings of IEEE Micro-30*, Piscataway, NJ, December 1997.

[5] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction*, June 1982.

[6] A. J. Goldberg and J. L. Hennessy. Mtool: An integrated system for performance debugging shared memory multiprocessor appliations. *IEEE Transactions on Parallel and Distributed Systems*, January 1993.

[7] John F. Reiser and Joseph P. Skudlarek. Program profiling problems, and a solution via machine language rewriting. *SIGPLAN Notices*, 29(1):37–45, January 1994.

[8] W. J. Conover. *Practical Nonparametric Statistics, 2ed.* John Wiley & Sons, New York 1980.

[9] Daniel A. Menasce, Vergilio A. F. Almeida, and Larry W. Dowdy. *Capacity Planning and Performance Modeling.* Prentice Hall, 1994.

[10] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *IEEE Computer*, pages 18–28, September 1991.

[11] Harold S. Stone and John Cocke. Computer architecture in the 1990s. *IEEE Computer*, pages 30–38, September 1991.

[12] Doug Burger, James R. Goodman, and Alain Kagi. Memory bandwidth limitations of future microprocessors. In *ISCA*, New York, NY, 1996.

[13] Sequent Computer Systems, Inc. *Guide to Parallel Programming*, 1988.

[14] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research.* Holden-Day, 1986.