SRC Technical Note
1997 - 016a

July 28, 1997
Modified September 3, 1997

Supersedes SRC Technical Note 1997 - 016

Continuous Profiling: Where Have All the Cycles Gone?

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

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

To appear in the ACM Transactions on Computer Systems. This paper is a slightly revised version of a paper that will also appear in the 16th ACM Symposium on Operating Systems Principles, October 5-8, 1997, St. Malo. France. Copyright ©1997 by ACM, Inc. All rights reserved. Republished by permission

Abstract

This paper describes the DIGITAL Continuous Profiling Infrastructure, a sampling-based profiling system designed to run continuously on production systems. The system supports multiprocessors, works on unmodified executables, and collects profiles for entire systems, including user programs, shared libraries, and the operating system kernel. Samples are collected at a high rate (over 5200 samples/sec per 333-MHz processor), yet with low overhead (1–3% slowdown for most workloads).

Analysis tools supplied with the profiling system use the sample data to produce a precise and accurate accounting, down to the level of pipeline stalls incurred by individual instructions, of where time is being spent. When instructions incur stalls, the tools identify possible reasons, such as cache misses, branch mispredictions, and functional unit contention. The fine-grained instruction-level analysis guides users and automated optimizers to the causes of performance problems and provides important insights for fixing them.

1 Introduction

The performance of programs running on modern high-performance computer systems is often hard to understand. Processor pipelines are complex, and memory system effects have a significant impact on performance. When a single program or an entire system does not perform as well as desired or expected, it can be difficult to pinpoint the reasons. The DIGITAL Continuous Profiling Infrastructure provides an efficient and accurate way of answering such questions.

The system consists of two parts, each with novel features: a data collection subsystem that samples program counters and records them in an on-disk database, and a suite of analysis tools that analyze the stored profile information at several levels, from the fraction of CPU time consumed by each program to the number of stall cycles for each individual instruction. The information produced by the analysis tools guides users to time-critical sections of code and explains in detail the static and dynamic delays incurred by each instruction.

We faced two major challenges in designing and implementing our profiling system: efficient data collection for a very high sampling rate, and the identification and classification of processor stalls from program-counter samples. The data collection system uses periodic interrupts generated by performance counters available on DIGITAL Alpha processors to sample program counter values. (Other processors, such as Intel's Pentium Pro and SGI's R10K, also have similar hardware support.) Profiles are collected for unmodified executables, and all code is profiled, including applications, shared libraries, device drivers, and the kernel. Thousands of samples are gathered each second, allowing useful profiles to be gathered in a relatively short time. Profiling is also efficient: overhead is about 1-3% of the processor time, depending on the workload. This permits the profiling system to be run continuously on production systems and improves the quality of the profiles by minimizing the perturbation of the system induced by profiling.

The collected profiles contain time-biased samples of program counter values: the number of samples associated with a particular program counter value is proportional to the total time spent executing that instruction. Samples that show the relative number of cache misses, branch mispredictions, etc. incurred by individual instructions may also be collected if the processor's performance counters support such events.

Some of the analysis tools use the collected samples to generate the usual histograms of time spent per image, per procedure, per source line, or per instruction. Other analysis tools use a detailed machine model and heuristics described in Section 6 to convert time-biased samples into the average number of cycles spent executing each instruction, the number of times each instruction was executed, and possible explanations for any static or dynamic stalls. Our techniques can deduce this information entirely from the time-biased program counter profiles and the binary executable, although the other types of samples, if available, may also be used to improve the accuracy of the results.

Section 3 contains several examples of the output from our tools. As discussed there, the combination of fine-grained instruction-level analysis and detailed profiling of long-running workloads has produced insights into performance that are difficult to achieve with other

This work was done at DIGITAL's Systems Research Center (SRC) and Western Research Laboratory (WRL). Anderson and Dean are at WRL, Sites is now at Adobe, and the remaining authors are at SRC. Sites may be reached at dsites@adobe.com; the other authors may be reached at {jennifer, berc, jdean, sanjay, monika, sleung, mtv, caw, weihl}@pa.dec.com. Inquiries about the system described in this paper should be sent to dcpi@pa.dec.com; more information, including the profiling system software, can also be found on the Web at http://www.research.digital.com/SRC/dcpi.

tools. These insights have been used to improve the performance of several major commercial applications.

The output of the analysis tools can be used directly by programmers; it can also be fed into compilers, linkers, post-linkers, and run-time optimization tools. The profiling system is freely available on the Web [7]; it has been running on DIGITAL Alpha processors under DIGITAL Unix since September 1996, and ports are in progress to Alpha/NT and OpenVMS. Work is underway to feed the output of our tools into DIGITAL's optimizing backend [3] and into the Spike/OM post-linker optimization framework [5, 6]. We are also studying new kinds of profile-driven optimizations made possible by the fine-grained instruction-level profile information provided by our system.

Section 2 discusses other profiling systems. Section 3 illustrates the use of our system. Sections 4 and 5 describe the design and performance of our data collection system, highlighting the techniques used to achieve low overhead with a high sampling rate. Section 6 describes the subtle and interesting techniques used in our analysis tools, explaining how to derive each instruction's CPI, execution frequency, and explanations for stalls from the raw sample counts. Finally, Section 7 discusses future work and Section 8 summarizes our results.

2 Related Work

Few other profiling systems can monitor complete system activity with high-frequency sampling and low overhead; only ours and *Morph* [26] are designed to run continuously for long periods on production systems, something that is essential for obtaining useful profiles of large complex applications such as databases. In addition, we know of no other system that can analyze time-biased samples to produce accurate fine-grained information about the number of cycles taken by each instruction and the reasons for stalls; the only other tools that can produce similar information use simulators, at much higher cost.

Table 1 compares several profiling systems. The *overhead* column describes how much profiling slows down the target program; low overhead is defined arbitrarily as less than 20%. The *scope* column shows whether the profiling system is restricted to a single application (App) or can measure full system activity (Sys). The *grain* column indicates the range over which an individual measurement applies. For example, *gprof* counts procedure executions, whereas *pixie* can count executions of each instruction. *prof* goes even further and reports the time spent executing each instruction, which, given the wide variations in latencies of different instructions, is often more useful than just an execution count. The *stalls* column indicates whether and how well the system can subdivide the time spent at an instruction into components like cache miss latency, branch misprediction delays, etc.

System	Overhead	Scope	Grain	Stalls
pixie	High	App	inst count	none
gprof	High	App	proc count	none
jprof	High	App	proc count	none
quartz	High	App	proc count	none
MTOOL	High	App	inst count/time	inaccurate
SimOS	High	Sys	inst time	accurate
SpeedShop (pixie)	High	App	inst count	none
VTune (dynamic)	High	App	inst time	accurate
prof	Low	App	inst time	none
iprobe	High	Sys	inst time	inaccurate
Morph	Low	Sys	inst time	none
VTune (sampler)	Low	Sys	inst time	inaccurate
SpeedShop (timer and counters)	Low	Sys	inst time	inaccurate
DCPI	Low	Sys	inst time	accurate

Table 1: Profiling systems

The systems fall into two groups. The first includes *pixie* [17], *gprof* [11], *jprof* [19], *quartz* [1], *MTOOL* [10], *SimOS* [20], part of SGI's *SpeedShop* [25], and Intel's *VTune* dynamic analyzer [24]. These systems use binary modification, compiler support, or direct simulation of programs to gather measurements. They all have high overhead and usually require significant user intervention. The slowdown is too large for continuous measurements during production use, despite techniques that reduce instrumentation overhead substantially [2]. In addition, only the simulation-based systems provide accurate information about the locations and causes of stalls.

The systems in the second group use statistical sampling to collect fine-grained information on program or system behavior. Some sampling systems, including *Morph* [26], *prof* [18], and part of *SpeedShop*, rely on an existing source of interrupts (e.g., timer interrupts) to generate program-counter samples. This prevents them from sampling within those interrupt routines, and can also result in correlations between the sampling and other system activity. By using hardware performance counters and randomizing the interval between samples, we are able to sample activity within essentially the entire system (except for our interrupt handler itself) and to avoid correlations with any other activity. This issue is discussed further in Section 4.1.1.

Other systems that use performance counters, including *iprobe* [13], the *VTune* sampler [24], and part of *SpeedShop*, share some of the characteristics of our system. However, *iprobe* and *VTune* cannot be used for continuous profiling, mostly because they need a lot of memory for sample data. In addition, *iprobe*, the *VTune* sampler, and *SpeedShop* all are unable to map the sample data accurately back to individual instructions. In contrast, our tools produce an accurate accounting of stall cycles incurred by each instruction and the potential reason(s) for the stalls.

3 Data Analysis Examples

To illustrate the range of information our system can provide, this section provides several examples of its use. Our system has been used to analyze and improve the performance of a wide range of complex commercial applications, including graphics systems, databases, industry benchmark suites, and compilers. For example, our tools pinpointed a performance problem in a commercial database system; fixing the problem reduced the response time of a particular SQL query from 180 to 14 hours. In another example, our tools' fine-grained instruction-level analyses identified opportunities to improve optimized code produced by DIGITAL's compiler, speeding up the mgrid SPECfp95 benchmark by 15%.

Our system includes a large suite of tools to analyze profiles at different levels of detail. In this section, we present several examples of the following tools:

- **dcpiprof**: Display the number of samples per procedure (or per image).
- **dcpicalc**: Calculate the cycles-per-instruction and basic block execution frequencies of a procedure, and show possible causes for stalls (see Section 6).
- **dcpistats**: Analyze the variations in profile data from many runs.

Other tools annotate source and assembly code with sample counts, highlight the differences in two separate profiles for the same program, summarize where time is spent in an entire program (the percentage of cycles spent waiting for data-cache misses, etc.; see Figure 4 for an example of this kind of summary for a single procedure), translate profile data into pixie format, and produce formatted Postscript output of annotated control-flow graphs.

3.1 Procedure-Level Bottlenecks

Dcpiprof provides a high-level view of the performance of a workload. It reads the profile data gathered by the system and displays a listing of the number of samples per procedure, sorted by decreasing number of samples. (It can also list the samples by image, rather than by procedure.) Figure 1 shows the first few lines of the output of dcpiprof for a run of `x11perf`, an X11 drawing benchmark. For example, the `ffb8ZeroPolyArc` routine accounts for 33.87% of the cycles for this workload. Notice that this profile includes code in the kernel (`/vmunix`) as well as code in shared libraries. The figure also has columns for the cumulative percent of cycle samples consumed by the procedure and all those preceding it in the listing, as well as information about the total number and fraction of instruction cache miss samples that occurred in each procedure.

3.2 Instruction-Level Bottlenecks

Dcpicalc provides a detailed view of the time spent on each instruction in a procedure. Figure 2 illustrates the output of dcpicalc for the key basic block in a McCalpin-like copy benchmark [15], running on an AlphaStation 500 5/333. The copy benchmark runs the following loop where `n` = 2000000 and the array elements are 64-bit integers:

```
for (i = 0; i < n; i++)
  c[i] = a[i];
```

The compiler has unrolled the loop four times, resulting in four loads and stores per iteration. The generated code shown in Figure 2 drives the memory system at full speed.

At the beginning of the basic block, dcpicalc shows summary information for the block. The first two lines display the best-case and actual cycles per instruction (CPI) for the block. The best-case scenario includes all stalls statically predictable from the instruction stream but assumes that there are no dynamic stalls (*e.g.*, all load instructions hit in the D-cache). For the copy benchmark, we see that the actual CPI is quite high at 10.77, whereas the best theoretical CPI (if no dynamic stalls occurred) is only 0.62. This shows that dynamic stalls are the significant performance problem for this basic block.

Dcpicalc also lists the instructions in the basic block, annotated with information about the stall cycles (and

```
Total samples for event type cycles = 6095201, imiss = 1117002

The counts given below are the number of samples for each listed event type.
==========================================================================================
 cycles       %    cum%   imiss       % procedure              image
2064143  33.87%  33.87%   43443   3.89% ffb8ZeroPolyArc        /usr/shlib/X11/lib_dec_ffb_ev5.so
 517464   8.49%  42.35%   86621   7.75% ReadRequestFromClient  /usr/shlib/X11/libos.so
 305072   5.01%  47.36%   18108   1.62% miCreateETandAET       /usr/shlib/X11/libmi.so
 271158   4.45%  51.81%   26479   2.37% miZeroArcSetup         /usr/shlib/X11/libmi.so
 245450   4.03%  55.84%   11954   1.07% bcopy                  /vmunix
 209835   3.44%  59.28%   12063   1.08% Dispatch               /usr/shlib/X11/libdix.so
 186413   3.06%  62.34%   36170   3.24% ffb8FillPolygon        /usr/shlib/X11/lib_dec_ffb_ev5.so
 170723   2.80%  65.14%   20243   1.81% in_checksum            /vmunix
 161326   2.65%  67.78%    4891   0.44% miInsertEdgeInET       /usr/shlib/X11/libmi.so
 133768   2.19%  69.98%    1546   0.14% miX1Y1X2Y2InRegion     /usr/shlib/X11/libmi.so
```

Figure 1: The key procedures from an x11perf run.

program source code, if the image contains line number information). Above each assembly instruction that stalls, dcpicalc inserts *bubbles* to show the duration and possible cause of the stall. Each line of assembly code shows, from left to right, the instruction's address, the instruction, the number of PC samples at this instruction, the average number of cycles this instruction spent at the head of the issue queue (stalled or not), and the addresses of other instructions that may have caused this instruction to stall. Note that Alpha load and load-address instructions write their first operand; 3-register operators write their third operand.

Each line in the listing represents a half-cycle, so it is easy to see if instructions are being dual-issued. In the figure, we see that there are two large stalls, one for 18.0 cycles at instruction 009828, and another for 114.5 cycles at instruction 009834. The letters `dwD` before the stalled `stq` instruction at 009828 indicate three possible reasons: a D-cache miss incurred by the `ldq` at 009810 (which provides the data needed by the `stq`), a write-buffer overflow, or a data TLB (DTB) miss. The `stq` instruction at 009834 is also stalled for the same three possible reasons. The lines labeled `s` indicate static stalls due to slotting hazards; in this case they are caused by the 21164 not being able to dual-issue adjacent `stq` instructions. Dcpicalc identifies these reasons by analyzing the instructions and the time-biased program counter samples, without monitoring other events like cache misses.

As expected, the listing shows that as the copy loop streams through the data the performance bottleneck is mostly due to memory latency. Also, the six-entry write buffer on the 21164 is not able to retire the writes fast enough to keep up with the computation. DTB miss is perhaps not a real problem since the loop walks through each page and may incur DTB misses only when crossing a page boundary. It would have been ruled out if samples for DTB miss events had been collected. Since they are not in this particular experiment (they are not collected by default), dcpicalc lists DTB miss as a possibility because it is designed to assume the worst unless the data indicate otherwise. Section 6.3 discusses this further.

3.3 Analyzing Variance Across Program Executions

Several benchmarks that we used to analyze the performance of the data collection system showed a no-

```
*** Best-case 8/13 =   0.62CPI
*** Actual  140/13 =  10.77CPI

Addr   Instruction        Samples   CPI  Culprit
    pD    (p = branch mispredict)
    pD    (D = DTB miss)
009810 ldq    t4, 0(t1)      3126   2.0cy
009814 addq   t0, 0x4, t0       0   (dual issue)
009818 ldq    t5, 8(t1)      1636   1.0cy
00981c ldq    t6, 16(t1)      390   0.5cy
009820 ldq    a0, 24(t1)     1482   1.0cy
009824 lda    t1, 32(t1)        0   (dual issue)
    dwD   (d = D-cache miss)
    dwD   ... 18.0cy
    dwD   (w = write-buffer overflow)
009828 stq    t4, 0(t2)     27766  18.0cy 9810
00982c cmpult t0, v0, t4        0   (dual issue)
009830 stq    t5, 8(t2)      1493   1.0cy
  s       (s = slotting hazard)
    dwD
    dwD ... 114.5cy
    dwD
009834 stq    t6, 16(t2)   174727 114.5cy 981c
  s
009838 stq    a0, 24(t2)     1548   1.0cy
00983c lda    t2, 32(t2)        0   (dual issue)
009840 bne    t4, 0x009810   1586   1.0cy
```

Figure 2: Analysis of Copy Loop.

```
Number of samples of type cycles
set 1 = 860301   set 2 = 862645   set 3 = 871952   set 4 = 870780          TOTAL      7144601
set 5 = 942929   set 6 = 893154   set 7 = 890969   set 8 = 951871

Statistics calculated using the sample counts for each procedure from 8 different sample set(s)
=============================================================================================
  range%        sum      sum%     N       mean    std-dev        min        max     procedure
  11.32%  441040.00     6.17%     8   55130.00   21168.70   38155.00   88075.00     smooth_
   1.44%   72385.00     1.01%     8    9048.12     368.74    8578.00    9622.00     fftb_
   1.39%   71129.00     1.00%     8    8891.12     327.68    8467.00    9453.00     fftf_
   0.94% 4242079.00    59.37%     8  530259.87   14097.11  515253.00  555180.00     parmvr_
   0.68%  378622.00     5.30%     8   47327.75    1032.09   46206.00   48786.00     putb_
   0.65%  410929.00     5.75%     8   51366.13    1161.61   50420.00   53110.00     vslv1p_
```

Figure 3: Statistics across eight runs of the SPECfp95 benchmark wave5.

ticeable variance in running times across different runs. We used our tools to examine one of these benchmarks, wave5 from the sequential SPECfp95 workload, in more detail.

We ran wave5 on an AlphaStation 500 5/333 and observed running times that varied by as much as 11%.

```
*** Best-case 14686/36016 =    0.41CPI,
*** Actual 35171/36016 =    0.98CPI
***
***  I-cache (not ITB)   0.0% to  0.3%
***   ITB/I-cache miss   0.0% to  0.0%
***       D-cache miss  27.9% to 27.9%
***           DTB miss   9.2% to 18.3%
***       Write buffer   0.0% to  6.3%
***    Synchronization   0.0% to  0.0%
***
***  Branch mispredict   0.0% to  2.6%
***          IMUL busy   0.0% to  0.0%
***          FDIV busy   0.0% to  0.0%
***              Other   0.0% to  0.0%
***
***  Unexplained stall   2.3% to  2.3%
***   Unexplained gain  -4.3% to -4.3%
*** ----------------------------------------
***   Subtotal dynamic                 44.1%
***
***           Slotting       1.8%
***      Ra dependency       2.0%
***      Rb dependency       1.0%
***      Rc dependency       0.0%
***      FU dependency       0.0%
*** ----------------------------------------
***    Subtotal static                  4.8%
*** ----------------------------------------
***        Total stall                 48.9%
***          Execution                 51.2%
*** Net sampling error                 -0.1%
*** ----------------------------------------
***      Total tallied                100.0%
*** (35171, 93.1% of all samples)
```

Figure 4: Summary of how cycles are spent in the procedure `smooth` for the fast run of the SPECfp95 benchmark wave5.

We ran dcpistats on 8 sets of sample files to isolate the procedures that had the greatest variance; dcpistats reads multiple sets of sample files and computes statistics comparing the profile data in the different sets. The output of dcpistats for wave5 is shown in Figure 3.

The figure shows the procedures in the wave5 program, sorted by the normalized range, *i.e.*, the difference between the maximum and minimum sample counts for that procedure, divided by the sum of the samples. We see that the procedure `smooth` had a much larger range than any of the other procedures.

Next, we ran dcpicalc on `smooth` for each profile, obtaining a summary of the fraction of cycles consumed by each type of dynamic and static stall within the procedure. The summary for the fastest run (the profile with the fewest samples) is shown in Figure 4. The summary for the slowest run (not shown) shows that the percentages of stall cycles attributed to D-cache miss, DTB miss, and write buffer overflow increase dramatically to 44.8-44.9%, 14.0-33.9%, and 0.0-18.3% respectively. The increase is probably in part due to differences in the virtual-to-physical page mapping across the different runs—if different data items are located on pages that map to the same location in the physically-addressed board cache (the L3 cache on the 21164), the number of conflict misses will increase.

4 Data Collection System

The analysis tools described in the previous section rely on profiles gathered as the workload executes. To gather these profiles, the DIGITAL Continuous Profiling Infrastructure periodically samples the program counter (PC) on each processor, associates each sample with its corresponding executable image, and saves the samples on disk in compact profiles. The key to our system's ability to support high-frequency continuous profiling is its efficiency: it uses about 1–3% of the CPU, and modest amounts of memory and disk. This is the direct result of careful design.

Sampling relies on the Alpha processor's performance-counter hardware to count various events, such as cycles and cache misses, for all instructions executed on the processor. Each processor generates a high-priority interrupt after a specified number of events has occurred, allowing the interrupted instruction and other context to be captured. Over time, the system gathers more and more samples, which provide an accurate statistical picture of the total number of events associated with each instruction in every executable image run on the system. (There are a few blind spots in uninterruptible code; however, all other code is profiled, unlike systems that rely on the real-time clock interrupt or other existing system functions to obtain samples.) The accumulated samples can then be analyzed, as discussed in Section 6, to reveal useful performance metrics at various levels of abstraction, including execution counts and the average number of stall cycles for each instruction, as shown in Section 3.

Figure 5 shows an overview of the data collection system. At an abstract level, the system consists of three interacting components: a kernel *device driver* that services performance-counter interrupts; a user-mode *daemon process* that extracts samples from the driver, associates them with executable images, and merges them into a nonvolatile profile database; and a *modified system loader* and other mechanisms for identifying executable images and where they are loaded by each running process. The rest of this section describes these pieces in more detail, beginning with the hardware performance counters.

4.1 Alpha Performance Counters

Alpha processors [9, 8] provide a small set of hardware performance counters that can each be configured to count a specified event. The precise number of counters, set of supported events, and other interface details vary across Alpha processor implementations. However, all existing Alpha processors can count a wide range of interesting events, including processor clock cycles (CYCLES), instruction cache misses (IMISS), data cache misses (DMISS), and branch mispredictions (BRANCHMP).

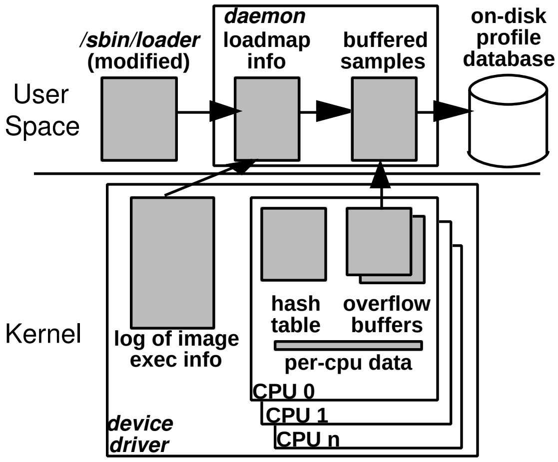

Figure 5: Data Collection System Overview

When a performance counter overflows, it generates a high-priority interrupt that delivers the PC of the next instruction to be executed [21, 8] and the identity of the overflowing counter. When the device driver handles this interrupt, it records the process identifier (PID) of the interrupted process, the PC delivered by the interrupt, and the event type that caused the interrupt.

Our system's default configuration monitors CYCLES and IMISS events.[1] Monitoring CYCLES results in periodic samples of the program counter, showing the total time spent on each instruction. Monitoring IMISS events reveals the number of times each instruction misses in the instruction cache. Our system can also be configured to monitor other events (*e.g.*, DMISS and BRANCHMP), giving more detailed information about the causes for dynamic stalls. Since only a limited number of events can be monitored simultaneously (2 on the 21064 and 3 on the 21164), our system also supports time-multiplexing among different events at a very fine grain. (SGI's Speedshop [25] provides a similar multiplexing capability.)

4.1.1 Sampling Period

Performance counters can be configured to overflow at different values; legal settings vary on different Alpha processors. When monitoring CYCLES on the Alpha 21064, interrupts can be generated every 64K events or every 4K events. On the 21164, each 16-bit performance counter register is writable, allowing any inter-interrupt period up to the maximum of 64K events to

[1] We monitor CYCLES to obtain the information needed to estimate instruction frequency and cpi; see Section 6 for details. We also monitor IMISS because the IMISS samples are usually accurate, so they provide important additional information for understanding the causes of stalls; see the discussion in Section 4.1.2.

be chosen. To minimize any systematic correlation between the timing of the interrupts and the code being run, we randomize the length of the sampling period by writing a pseudo-random value [4] into the performance counter at the end of each interrupt. The default sampling period is distributed uniformly between 60K and 64K when monitoring CYCLES.

4.1.2 Attributing Events to PCs

To accurately interpret samples, it is important to understand the PC delivered to the interrupt handler. On the 21164, a performance counter interrupt is delivered to the processor exactly six cycles after the counter overflows. When the interrupt is delivered, the handler is invoked with the PC of the oldest instruction that was in the issue queue at the time of interrupt delivery. The delayed delivery does not skew the distribution of cycle counter overflows; it just shifts the sampling period by six cycles. The number of cycle counter samples associated with each instruction is still statistically proportional to the total time spent by that instruction at the head of the issue queue. Since instructions stall only at the head of the issue queue on the 21064 and 21164, this accounts for all occurrences of stalls.

Events that incur more than six cycles of latency can mask the interrupt latency. For example, instruction-cache misses usually take long enough that the interrupt is delivered to the processor before the instruction that incurred the IMISS has issued. Thus, the sampled PC for an IMISS event is usually (though not always) correctly attributed to the instruction that caused the miss.

For other events, the six-cycle interrupt latency can cause significant problems. The samples associated with events caused by a given instruction can show up on instructions a few cycles later in the instruction stream, depending on the latency of the specific event type. Since a dynamically varying number of instructions, including branches, can occur during this interval, useful information may be lost. In general, samples for events other than CYCLES and IMISS are helpful in tracking down performance problems, but less useful for detailed analysis.

4.1.3 Blind Spots: Deferred Interrupts

Performance-counter interrupts execute at the highest kernel priority level (`spldevrt`), but are deferred while running non-interruptible PALcode [21] or system code at the highest priority level.[2] Events in PALcode and high-priority interrupt code are still counted, but samples for those events will be associated with the instruction that runs after the PALcode finishes or the interrupt level drops below `spldevrt`.

For synchronous PAL calls, the samples attributed to the instruction following the call provide useful information about the time spent in the call. The primary asynchronous PAL call is "deliver interrupt," which dispatches to a particular kernel entry point; the samples for "deliver interrupt" accumulate at that entry point. The other samples for high-priority asynchronous PAL calls and interrupts are both relatively infrequent and usually spread throughout the running workload, so they simply add a small amount of noise to the statistical sampling.

4.2 Device Driver

Our device driver efficiently handles interrupts generated by Alpha performance counter overflows, and provides an `ioctl` interface that allows user-mode programs to flush samples from kernel buffers to user space.

The interrupt rate is high: approximately 5200 interrupts per second on each processor when monitoring CYCLES on an Alpha 21164 running at 333 MHz, and higher with simultaneous monitoring of additional events. This raises two problems. First, the interrupt handler has to be fast; for example, if the interrupt handler takes 1000 cycles, it will consume more than 1.5% of the CPU. Note that a cache miss all the way to memory costs on the order of 100 cycles; thus, we can afford to execute lots of instructions but not to take many cache misses. Second, the samples generate significant memory traffic. Simply storing the raw data (16-bit PID, 64-bit PC, and 2-bit EVENT) for each interrupt in a buffer would generate more than 52 KB per processor per second. This data will be copied to a user-level process for further processing and merging into on-disk profiles, imposing unacceptable overhead.

We could reduce these problems by resorting to lower-frequency event sampling, but that would increase the amount of time required to collect useful profiles. Instead, we engineered our data collection system to reduce the overhead associated with processing each sample. First, we reduce the number of samples that have to be copied to user space and processed by the

[2] This makes profiling the performance-counter interrupt handler difficult. We have implemented a "meta" method for obtaining samples within the interrupt handler itself, but space limitations preclude a more detailed discussion.

daemon by counting, in the device driver, the number of times a particular sample has occurred recently. This typically reduces the data rate of sample data moving from the device driver to the user-level daemon by a factor of 20 or more. Second, we organize our data structures to minimize cache misses. Third, we allocate per-processor data structures to reduce both writes to shared cache lines and the synchronization required for correct operation on a multiprocessor. Fourth, we switch dynamically among specialized versions of the interrupt handler to reduce the time spent checking various flags and run-time constants. The rest of this section describes our optimizations in more detail.

4.2.1 Data Structures

Each processor maintains its own private set of data structures. A processor's data structures are primarily modified by the interrupt routine running on that processor. However, they can also be read and modified by the flush routines that copy data to user space. Synchronization details for these interactions are discussed in Section 4.2.3.

Each processor maintains a *hash table* that is used to aggregate samples by counting the number of times each (PID, PC, EVENT) triple has been seen. This reduces the amount of data that must be passed from the device driver to the user-level daemon by a factor of 20 or more for most workloads, resulting in less memory traffic and lower processing overhead per aggregated sample. The hash table is implemented with an array of fixed size buckets, where each bucket can store four entries (each entry consists of a PID, PC, and EVENT, plus a count).

A pair of *overflow buffers* stores entries evicted from the hash table. Two buffers are kept so entries can be appended to one while the other is copied to user space. When an overflow buffer is full, the driver notifies the daemon, which copies the buffer to user space.

The interrupt handler hashes the PID, PC, and EVENT to obtain a bucket index i; it then checks all entries at index i. If one matches the sample, its count is incremented. Otherwise one entry is evicted to an overflow buffer and is replaced by the new sample with a count of one. The evicted entry is chosen using a mod-4 counter that is incremented on each eviction. Each entry occupies 16 bytes; therefore, a bucket occupies one cache line (64 bytes) on an Alpha 21164, so we incur at most one data-cache miss to search the entire bucket.

The four-way associativity of the hash table helps to prevent thrashing of entries due to hashing collisions. In Section 5 we discuss experiments conducted to evaluate how much greater associativity might help.

4.2.2 Reducing Cache Misses

A cache miss all the way out to memory costs on the order of 100 cycles. Indeed, it turns out that cache misses, for both instructions and data, are one of the dominant sources of overhead in the interrupt handler; we could execute many more instructions without a significant impact on overhead as long as they did not result in cache misses.

To reduce overhead, we designed our system to minimize the number of cache misses. In the common case of a hash table hit, the interrupt handler accesses one bucket of the hash table; various private per-processor state variables such as a pointer to the local hash table, the seed used for period randomization, etc; and global state variables such as the size of the hash table, the set of monitored events, and the sampling period.

On the 21164, the hash table search generates at most one cache miss. Additionally, we pack the private state variables and read-only copies of the global variables into a 64 byte per-processor data structure, so at most one cache miss is needed for them. By making copies of all shared state, we also avoid interprocessor cache line thrashing and invalidations.

In the uncommon case of a hash table miss, we evict an old entry from the hash table. This eviction accesses one extra cache line for the empty overflow buffer entry into which the evicted entry is written. Some per-processor and global variables are also accessed, but these are all packed into the 64 byte per-processor structure described above. Therefore these accesses do not generate any more cache misses.

4.2.3 Reducing Synchronization

Synchronization is eliminated between interrupt handlers on different processors in a multiprocessor, and minimized between the handlers and other driver routines. Synchronization operations (in particular, memory barriers [21]) are expensive, costing on the order of 100 cycles, so even a small number of them in the interrupt handler would result in unacceptable overhead. The data structures used by the driver and the techniques used to synchronize access to them were designed to eliminate *all* expensive synchronization operations from the interrupt handler.

We use a separate hash table and pair of overflow buffers per processor, so handlers running on different processors never need to synchronize with each other.

Synchronization is only required between a handler and the routines that copy the contents of the hash table and overflow buffers used by that handler to user space. Each processor's hash table is protected by a flag that can be set only on that processor. Before a flush routine copies the hash table for a processor, it performs an inter-processor interrupt (IPI) to that processor to set the flag indicating that the hash table is being flushed. The IPI handler raises its priority level to ensure that it executes atomically with respect to the performance-counter interrupts. If the hash table is being flushed, the performance counter interrupt handler writes the sample directly into the overflow buffer. Use of the overflow buffers is synchronized similarly.

Although IPIs are expensive, they allow us to remove all memory barriers from the interrupt handler, in exchange for increasing the cost of the flush routines. Since the interrupt handler runs much more frequently than the flush routines, this is a good tradeoff.

4.3 User-Mode Daemon

A user-mode daemon extracts samples from the driver and associates them with their corresponding images. Users may also request separate, per-process profiles for specified images. The data for each image is periodically merged into compact profiles stored as separate files on disk.

4.3.1 Sample Processing

The main daemon loop waits until the driver signals a full overflow buffer; it then copies the buffer to user space and processes each entry. The daemon maintains image maps for each active process; it uses the PID and the PC of the entry to find the image loaded at that PC in that process. The PC is converted to an image offset, and the result is merged into a hash table associated with the relevant image and EVENT. The daemon obtains its information about image mappings from a variety of sources, as described in the following section.

Periodically, the daemon extracts all samples from the driver data structures, updates disk-based profiles and discards data structures associated with terminated processes. The time intervals associated with periodic processing are user-specified parameters; by default, the daemon drains the driver every 5 minutes, and in-memory profile data is merged to disk every 10 minutes. This simple timeout-based approach can cause undesirable bursts of intense daemon activity; the next version of our system will avoid this by updating disk profiles incrementally. A complete flush can also be initiated by a user-level command.

4.3.2 Obtaining Image Mappings

We use several sources of information to determine where images are loaded into each process. First, a modified version of the dynamic system loader (`/sbin/loader`) notifies our system's daemon whenever an image is loaded into a process. The notification contains the PID, a unique identifier for each loaded image, the address at which it was loaded, and its filesystem pathname. This mechanism captures all dynamically loaded images.

Second, the kernel `exec` path invokes a chain of recognizer routines to determine how to load an image. We register a special routine at the head of this chain that captures information about all static images. The recognizer stores this data in a kernel buffer that is flushed by the daemon every few seconds.

Finally, to obtain image maps for processes already active when the daemon starts, on start-up the daemon scans all active processes and their mapped regions using Mach-based system calls available in DIGITAL Unix.

Together, these mechanisms are able to successfully classify virtually all samples collected by the driver. Any remaining unknown samples are aggregated into a special profile. In our experience, the number of unknown samples is considerably smaller than 1%; a typical fraction from a week-long run is 0.05%.

4.3.3 Profile Database

The daemon stores samples in an on-disk profile database. This database resides in a user-specified directory, and may be shared by multiple machines over a network. Samples are organized into non-overlapping *epochs*, each of which contains all samples collected during a given time interval. A new epoch can be initiated by a user-level command. Each epoch occupies a separate sub-directory of the database. A separate file is used to store the profile for a given image and EVENT combination.

The profile files are written in a compact binary format. Since significant fractions of most executable images consist of symbol tables and instructions that are never executed, profiles are typically smaller than their associated executables by an order of magnitude, even after days of continuous profiling. Although disk space usage has not been a problem, we have also designed an improved format that can compress existing profiles by approximately a factor of three.

Workload	Mean *base* runtime (secs)	Platform	Description
Uniprocessor workloads			
SPECint95	13226 ± 258	333 MHz ALPHASTATION 500	The SPEC benchmark suite compiled using both the BASE and PEAK compilation flags and run with the *runspec* driver [22].
SPECfp95	17238 ±106	333 MHz ALPHASTATION 500	
x11perf	N/A	333 MHz ALPHASTATION 500	Several tests from the x11perf X server performance testing program. The tests chosen are representative of CPU-bound tests [16].
McCalpin	N/A	333 MHz ALPHASTATION 500	The McCalpin STREAMS benchmark, consisting of four loops that measure memory-system bandwidth [15].
Multiprocessor workloads			
AltaVista	319 ± 2	300 MHz 4-CPU ALPHASERVER 4100	A trace of 28622 queries made to the 3.5 GB AltaVista news index. The system was driven so as to maintain 8 outstanding queries.
DSS	2786 ± 35	300 MHz 8-CPU ALPHASERVER 8400	A decision-support system (DSS) query based upon the TPC-D specification [23].
parallel SPECfp	2777 ± 168	300 MHz 4-CPU ALPHASERVER 4100	The SPECfp95 programs, parallelized by the Stanford SUIF compiler [12].
timesharing	7 days	300 MHz 4-CPU ALPHASERVER 4100	A timeshared server used for office and technical applications, running the *default* configuration of our system. We used this workload to gather statistics for a long-running profile session.

Table 2: Description of Workloads

5 Profiling Performance

Performance is critical to the success of a profiling system intended to run continuously on production systems. The system must collect many thousands of samples per second yet incur sufficiently low overhead that its benefits outweigh its costs. In this section we summarize the results of experiments designed to measure the performance of our system and to explore tradeoffs in its design.

We evaluated our profiling system's performance under three different configurations: *cycles*, in which the system monitors only cycles, *default*, in which the system monitors both cycles and instruction-cache misses, and *mux*, in which the system monitors cycles with one performance counter and uses multiplexing to monitor instruction-cache misses, data-cache misses, and branch mispredictions with another counter. Table 2 shows the workloads used, their average running times (from a minimum of 10 runs, shown with 95%-confidence intervals) in the *base* configuration without our system, and the machines on which they ran.

5.1 Aggregate Time Overhead

To measure the overhead, we ran each workload a minimum of 10 times in each configuration, and ran many workloads as many as 50 times. Table 3 shows the percentage overhead (with 95%-confidence intervals) imposed by the three different configurations of our system compared to the *base* configuration. (The timesharing workload is not included in the table; since it was measured on a live system, we cannot run it in each configuration to determine overall slowdown.) McCalpin and x11perf report their results as rates (MB/sec for McCalpin, and operations/sec for x11perf); for these, the table shows the degradation of the rates. For the other workloads, the table shows the increase in running time. The numbers in Table 3 show that the overall overhead imposed by our system is quite low, usually 1 to 3%. The variation in performance from run to run of each workload is typically much greater than our system's overhead.

Workload	*cycles (%)*	*default (%)*	*mux (%)*
Uniprocessor workloads			
SPECint95	2.0 ± 0.8	2.8 ± 0.9	3.0 ± 0.7
SPECfp95	0.6 ± 1.0	0.5 ± 1.1	1.1 ± 1.1
x11perf			
noop	1.6 ± 0.5	1.9 ± 0.5	2.2 ± 0.5
circle10	2.8 ± 0.6	2.4 ± 0.4	2.4 ± 0.4
ellipse10	1.5 ± 0.2	1.8 ± 0.2	2.3 ± 0.4
64poly10	1.1 ± 0.4	2.0 ± 0.5	2.4 ± 0.6
ucreate	2.7 ± 0.7	4.2 ± 0.7	5.0 ± 0.7
McCalpin			
assign	0.9 ± 0.1	0.9 ± 0.1	1.1 ± 0.1
saxpy	1.0 ± 0.1	1.1 ± 0.1	1.3 ± 0.1
scale	1.1 ± 0.1	1.1 ± 0.1	1.2 ± 0.1
sum	1.1 ± 0.1	1.1 ± 0.1	1.2 ± 0.1
Multiprocessor workloads			
AltaVista	0.5 ± 0.8	1.3 ± 1.8	1.6 ± 0.5
DSS	1.2 ± 1.1	1.8 ± 2.6	0.6 ± 0.3
parallel SPECfp	6.0 ± 3.5	3.1 ± 1.8	7.5 ± 4.6

Table 3: Overall Slowdown (in percent)

Figure 6 shows the data in more detail for three programs: AltaVista; the gcc portion of the SPECint95 workload (peak version); and the wave5 portion of the

SPECfp95 workload (peak version). Each graph gives a scatter plot of the running times in seconds for all four configurations. The range of the y-axis is from 90% to 135% of the mean value, with the x-axis intersecting at the mean value. 95%-confidence intervals are also shown.

AltaVista is representative of the majority of the workloads that we studied: the profiling overhead is small and there is little variance across the different runs. In contrast, our system incurs relatively high overhead on gcc (about 4% to 10%). This benchmark compiles 56 pre-processed source files into assembly files; each file requires a separate invocation of the program and thus has a distinct PID. Since samples with distinct PID's do not match in the hash table, the eviction rate is high, resulting in higher overhead (see section 5.2). Finally, the wave5 data shows an apparent speedup from running DCPI in our experiments. In this and similar cases, the running time variance exceeded our profiling overhead.

The overheads we measured are likely to be slightly higher than would be experienced in practice, since as discussed in the next section, all measurements were done using an instrumented version of the system that logged additional statistics, imposing overhead that would not normally be incurred.

5.2 Components of Time Overhead

There are two main components to our system's overhead. First is the time to service performance-counter interrupts. Second is the time to read samples from the device driver into the daemon and merge the samples into the on-disk profiles for the appropriate images. To investigate the cost of these two components, we performed all the experiments with our system instrumented to collect several statistics: (1) the number of cycles spent in our interrupt handler, collected separately for the cases when samples hit or miss in the hash table; (2) the eviction rate from the hash table; and (3) the total number of samples observed. For real workloads, we are able to directly measure only the time spent in our interrupt handler, which does not include the time to deliver the interrupt nor the time to return from the interrupt handler. Experimentation with a tight spin loop revealed the best-case interrupt setup and teardown time to be around 214 cycles (not including our interrupt handler itself). Under real workloads, this value is likely to increase due to additional instruction-cache misses.

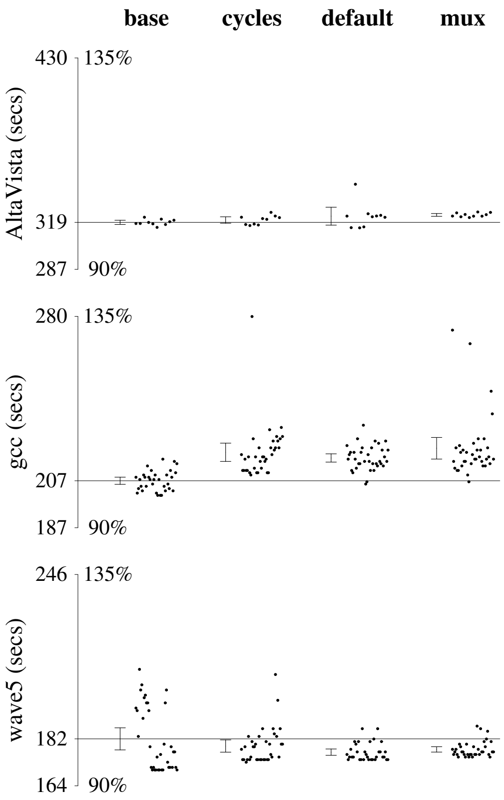

Figure 6: Distribution of running times

To evaluate the daemon's per-sample cost of processing, all experiments were configured to gather per-process samples for the daemon itself; this showed how many cycles were spent both in the daemon and in the kernel on behalf of the daemon. Dividing this by the total number of samples processed by the driver gives the per-sample processing time in the daemon.[3]

These statistics are summarized for each workload in Table 4 for each of the three configurations. We also separately measured the statistics for the gcc program in the SPECint95 workload to show the effects of a high eviction rate. The table shows that workloads with low eviction rates, such as SPECfp95 and AltaVista, not only spend less time processing each interrupt (because a hit in the hash table is faster than a miss), but also spend less time processing each sample in the daemon because many samples are aggregated into a single entry before being evicted from the hash table. For workloads with a high eviction rate, the average interrupt cost is higher; in addition, the higher eviction rate leads

[3]The per-sample metric is used to allow comparison with the per-sample time in the interrupt handler, and is different from the time spent processing each entry from the overflow buffer (since multiple samples are "processed" for entries with counts higher than one).

Workload	cycles				default				mux			
		per sample cost (cycles)				per sample cost (cycles)				per sample cost (cycles)		
	miss rate	intr cost avg (hit/miss)	daemon cost		miss rate	intr cost avg (hit/miss)	daemon cost		miss rate	intr cost avg (hit/miss)	daemon cost	
SPECint95	6.7%	435 (416/700)	175		9.5%	451 (430/654)	245		9.5%	582 (554/842)	272	
gcc	38.1%	551 (450/716)	781		44.5%	550 (455/669)	927		44.2%	667 (558/804)	982	
SPECfp95	0.6%	486 (483/924)	59		1.4%	437 (433/752)	95		1.5%	544 (539/883)	107	
x11perf	2.1%	464 (454/915)	178		5.6%	454 (436/763)	266		5.5%	567 (550/868)	289	
McCalpin	0.7%	388 (384/1033)	51		1.4%	391 (384/916)	70		1.1%	513 (506/1143)	72	
AltaVista	0.5%	343 (340/748)	21		1.7%	349 (344/661)	56		1.6%	387 (382/733)	47	
DSS	0.5%	230 (227/755)	41		0.9%	220 (216/660)	49		0.9%	278 (273/815)	60	
parallel SPECfp	0.3%	356 (354/847)	29		0.7%	355 (352/713)	47		0.9%	444 (440/854)	58	
timesharing	not measured				0.7%	202 (199/628)	66		not measured			

Table 4: Time overhead components

to more overflow entries and a higher per-sample cost in the daemon.

5.3 Aggregate Space Overhead

This section evaluates the memory and disk overheads of the system. Memory is consumed by both the device driver and the daemon, while disk space is used to store nonvolatile profile data.

As described in Section 4, the device driver maintains a hash table and a pair of overflow buffers for each processor in non-pageable kernel memory. In all of our experiments, each overflow buffer held 8K samples and each hash table held 16K samples, for a total of 512KB of kernel memory per processor.

The daemon consumes ordinary pageable memory. It allocates a buffer large enough to flush one overflow buffer or hash table per processor, as well as data structures for every active process and image. Memory usage grows with the number of active processes, and also depends upon workload locality. Per-process data structures are reaped infrequently (by default, every 5 minutes), and samples for each image are buffered until saved to disk (by default, every 10 minutes); as a result, the daemon's worst-case memory consumption occurs when the profiled workload consists of many short-lived processes or processes with poor locality.

Table 5 presents the average and peak resident memory (both text and data) used by the daemon for each workload. It also shows the length of time the daemon was up for running each workload in each configuration. For most workloads, memory usage is modest. The week-long timesharing workload, running on a four-processor compute server with hundreds of active processes, required the most memory (14.2 MB). However, since this multiprocessor has 4GB of physical memory, the overall fraction of memory devoted to our profiling system is less than 0.5%.

On workstations with smaller configurations (64MB to 128MB), the memory overhead ranges from 5 to 10%. Since the current daemon implementation has not been carefully tuned, we expect substantial memory savings from techniques such as reductions in the storage costs of hash tables and more aggressive reaping of inactive structures.

Finally, as shown in Table 5, the disk space consumed by profile databases is small. Most sets of profiles required only a few megabytes of storage. Even the week-long timesharing workload, which stored both CYCLES and IMISS profiles for over 480 distinct executable images, used just 13MB of disk space.

5.4 Potential Performance Improvements

While the driver has been carefully engineered for performance, there is still room for improvement. In addition, the performance of the daemon can probably be improved substantially.

As shown in Section 5.2, the performance of our system is heavily dependent on the effectiveness of the hash table in aggregating samples. To explore alternative designs, we constructed a trace-driven simulator that models the driver's hash table structures. Using sample traces logged by a special version of the driver, we examined varying associativity, replacement policy, overall table size and hash function.

Our experiments indicate that (1) increasing associativity from 4-way to 6-way, by packing more entries per processor cache line (which would also increase the total number of entries in the hash table), and (2) using swap-to-front on hash-table hits and inserting new entries at the beginning of the line, rather than the round-robin policy we currently use, would reduce the overall system cost by 10-20%. We intend to incorporate both of these changes in a future version of our system.

Unlike the driver, the user-mode daemon has not been heavily optimized. A few key changes should re-

Workload	*cycles*			*default*			*mux*		
	Uptime	Space (KBytes)		Uptime	Space (KBytes)		Uptime	Space (KBytes)	
		Memory avg (peak)	Disk usage		Memory avg (peak)	Disk usage		Memory avg (peak)	Disk usage
SPECint95	14:57:50	6600 (8666)	2639	15:00:36	8284 (13500)	4817	15:08:45	8804 (11250)	6280
gcc	5:49:37	8862 (11250)	1753	5:42:10	9284 (9945)	3151	5:47:44	11543 (12010)	4207
SPECfp95	19:15:20	2364 (3250)	1396	19:14:17	2687 (3750)	2581	19:22:37	2958 (3800)	3182
x11perf	0:21:25	1586 (1750)	216	0:20:58	1786 (1917)	356	0:21:31	1959 (2141)	434
McCalpin	0:09:10	1568 (2000)	108	0:09:07	1716 (2179)	155	0:09:09	1812 (2311)	157
AltaVista	0:26:49	2579 (3000)	265	0:27:04	2912 (3286)	470	0:27:09	3156 (3571)	571
DSS	3:55:14	4389 (5500)	634	3:56:23	5126 (5288)	1114	3:53:41	5063 (5242)	1389
parallel SPECfp	8:10:49	2902 (3250)	1157	7:57:02	3384 (3636)	2028	8:17:34	3662 (3950)	2616
timesharing	not measured			187:43:46	10887 (14200)	12601	not measured		

Table 5: Daemon Space Overhead

duce the time to process each raw driver sample significantly. One costly activity in the daemon involves associating a sample with its corresponding image; this currently requires three hash lookups. Sorting each buffer of raw samples by PID and PC could amortize these lookups over a large number of samples. Memory copy costs could also be reduced by mapping kernel sample buffers directly into the daemon's address space. We estimate that these and other changes could cut the overhead due to the daemon by about a factor of 2.

6 Data Analysis Overview

The CYCLES samples recorded by the data collection subsystem tell us approximately how much total time was spent by each instruction at the head of the issue queue. However, when we see a large sample count for an instruction, we do not know immediately from the sample counts whether the instruction was simply executed many times or whether it stalled most of the times it was executed. In addition, if the instruction did stall, we do not know why. The data analysis subsystem fills in these missing pieces of information. Note that the analysis is done offline, after samples have been collected.

Given profile data, the analysis subsystem produces for each instruction:

- A *frequency*, which is proportional to the number of times the instruction was executed during the profiled period;
- A *cpi*, which is an estimate of the average number of cycles spent by that instruction at the head of the issue queue for each execution during the profiled period; and
- A set of *culprits*, which are possible explanations for any wasted issue slots (due to static or dynamic stalls).

The analysis is done in two phases; the first phase estimates the frequency and cpi for each instruction, and the second phase identifies culprits for each stall. The analysis is designed for processors that execute instructions in order; we are working on extending it to out-of-order processors.

For programs whose executions are deterministic, it is possible to measure the execution counts by instrumenting the code directly (*e.g.*, using pixie). In this case, the first phase of the analysis, which estimates the frequency, is not necessary. However, many large systems (*e.g.*, databases) are not deterministic; even for deterministic programs, the ability to derive frequency estimates from sample counts eliminates the need to create and run an instrumented version of the program, simplifying the job of collecting profile information.

6.1 Estimating Frequency and CPI

The crux of the problem in estimating instruction frequency and cpi is that the sample data provides information about the total time spent by each instruction at the head of the issue queue, which is proportional to the product of its frequency and its cpi; we need to factor that product. For example, if the instruction's sample count is 1000, its frequency could be 1000 and its cpi 1, or its frequency could be 10 and its cpi 100; we cannot tell given only its sample count. However, by combining information from several instructions, we can often do an excellent job of factoring the total time spent by an instruction into its component factors.

The bulk of the estimation process is focused on estimating the frequency, F_i, of each instruction i. F_i is simply the number of times the instruction was executed divided by the average sampling period, P, used to gather the samples. The sample count S_i should be approximately F_iC_i, where C_i is the average number of cycles instruction i spends at the head of the issue queue. Our analysis first finds F_i; C_i is then easily obtained by division.

The analysis estimates the F_i values by examining one procedure at a time. The following steps are performed for each procedure:

1. Build a control-flow graph (CFG) for the procedure.
2. Group the basic blocks and edges of the CFG into equivalence classes based on frequency of execution.
3. Estimate the frequency of each equivalence class that contains instructions with suitable sample counts.
4. Use a linear-time local propagation method based on flow constraints in the procedure's CFG to propagate frequency estimates around the CFG.
5. Use a heuristic to predict the accuracy of the estimates.

Some details are given below.

6.1.1 Building a CFG

The CFG is built by extracting the code for a procedure from the executable image. Basic block boundaries are identified from instructions that change control flow, *e.g.*, branches and jumps. For indirect jumps, we analyze the preceding instructions to try to determine the possible targets of the jump. Sometimes this analysis fails, in which case the CFG is noted as missing edges. The current analysis does not identify interprocedural edges (*e.g.*, from calls to longjmp), nor does it note their absence.

6.1.2 Determining Frequency Equivalence

If the CFG is noted as missing edges, each block and each edge is assigned its own equivalence class. Otherwise, we use an extended version of the cycle equivalence algorithm in [14] to identify sets of blocks and edges that are guaranteed to be executed the same number of times. Each such set constitutes one equivalence class. Our extension to the algorithm is for handling CFG's with infinite loops, *e.g.*, the idle loop of an operating system.

6.1.3 Estimating Frequency From Sample Counts

The heuristic for estimating the frequency of an equivalence class of instructions works on one class at a time. All instructions in a class have the same frequency, henceforth called F.

```
Addr   Instruction               S_i M_i  S_i/M_i
009810 ldq    t4, 0(t1)         3126   1     3126
009814 addq   t0, 0x4, t0          0   0
009818 ldq    t5, 8(t1)         1636   1     1636
00981c ldq    t6, 16(t1)         390   0
009820 ldq    a0, 24(t1)        1482   1     1482 *
009824 lda    t1, 32(t1)           0   0
009828 stq    t4, 0(t2)        27766   1    27766
00982c cmpult t0, v0, t4           0   0
009830 stq    t5, 8(t2)         1493   1     1493 *
009834 stq    t6, 16(t2)      174727   1   174727
009838 stq    a0, 24(t2)        1548   1     1548 *
00983c lda    t2, 32(t2)           0   0
009840 bne    t4, 0x009810      1586   1     1586 *
```

Figure 7: Estimating Frequency of Copy Loop.

The heuristic is based on two assumptions: first, that at least some instructions in the class encounter no dynamic stalls, and second, that one can statically compute, for most instructions, the minimum number of cycles M_i that instruction i spends at the head of the issue queue in the absence of dynamic stalls.

M_i is obtained by scheduling each basic block using a model of the processor on which it was run. M_i may be 0. In practice, M_i is 0 for all but the first of a group of multi-issued instructions. An *issue point* is an instruction with $M_i > 0$.

If issue point i has no dynamic stalls, the frequency F should be, modulo sampling error, S_i/M_i. If the issue point incurs dynamic stalls, S_i will increase. Thus, we can estimate F by averaging some of the smaller ratios S_i/M_i of the issue points in the class.

As an example, Figure 7 illustrates the analysis for the copy loop shown previously in Figure 2. The M_i column shows the output from the instruction scheduler, and the S_i/M_i column shows the ratio for each issue point. The heuristic used various rules to choose the ratios marked with * to be averaged, computing a frequency of 1527. This is close to 1575.1, the true frequency for this example.

There are several challenges in making accurate estimates. First, an equivalence class might have few issue points. In general, the smaller the number of issue points, the greater the chance that all of them encounter some dynamic stall. In this case, the heuristic will overestimate F. At the extreme, a class might have no issue points, *e.g.*, because it contains no basic blocks. In this case, the best we can do is exploit flow constraints of the CFG to compute a frequency in the propagation phase.

Second, an equivalence class might have only a small number of samples. In this case, we estimate F as

$\sum_i S_i / \sum_i M_i$, where i ranges over the instructions in the class. This increases the number of samples used by our heuristic and generally improves the estimate.

Third, M_i may not be statically determinable. For example, the number of cycles an instruction spends at the head of the issue queue may in general depend on the code executed before the basic block. When a block has multiple predecessors, there is no one static code schedule for computing M_i. In this case, we currently ignore all preceding blocks. For the block listed in Figure 7, this limitation leads to an error: M_i for the `ldq` instruction at 009810 should be 2 instead of 1 because the processor cannot issue a `ldq` two cycles after the `stq` at 009838 from the previous iteration. Thus, a static stall was misclassified as a dynamic stall and the issue point was ignored.

Fourth, dynamic stalls sometimes make the M_i values inaccurate. Suppose an issue point instruction i depends on a preceding instruction j, either because i uses the result of j or because i needs to use some hardware resource also used by j. Thus, M_i is a function of the latency of j. If an instruction between j and i incurs a dynamic stall, this will cause i to spend fewer than M_i cycles at the head of the issue queue because the latency of j overlaps the dynamic stall. To address this problem, we use the ratio $\sum_{k=j+1}^{i} S_k / \sum_{k=j+1}^{i} M_k$ for the issue point i when there are instructions between j and i. This estimate is more reliable than S_i/M_i because the dependence of i on j ensures that the statically determined latency between them will not be decreased by dynamic stalls of j or intervening instructions.

Finally, one must select which of the ratios to include in the average. In rough terms, we examine clusters of issue points that have relatively small ratios, where a cluster is a set of issue points that have similar ratios (*e.g.*, maximum ratio in cluster $\leq 1.5 *$ minimum ratio in cluster). However, to reduce the chance of underestimating F, the cluster is discarded if its issue points appear to have anomalous values for S_i or M_i, *e.g.*, because the cluster contains less than a minimum fraction of the issue points in the class or because the estimate for F would imply an unreasonably large stall for another instruction in the class.

6.1.4 Local Propagation

Local propagation exploits flow constraints of the CFG to make additional estimates. Except for the boundary case where a block has no predecessors (or successors), the frequency of a block should be equal to the sum of the frequencies of its incoming (or outgoing) edges.

The flow constraints have the same form as dataflow equations, so for this analysis we use a variant of the standard, iterative algorithm used in compilers. The variations are (1) whenever a new estimate is made for a block or an edge, the estimate is immediately propagated to all of the other members in the block or edge's equivalence class, and (2) no negative estimates are allowed. (The flow equations can produce negative values because the frequency values are only estimates.) Because of the nature of the flow constraints, the time required for local propagation is linear in the size of the CFG.

We are currently experimenting with a global constraint solver to adjust the frequency estimates where they violate the flow constraints.

6.1.5 Predicting Accuracy of Estimates

The analysis uses a second heuristic to predict the accuracy of each frequency estimate as being *low*, *medium*, or *high confidence*. The confidence of an estimate is a function of the number of issue points used to compute the estimate, how tightly the ratios of the issue points were clustered, whether the estimate was made by propagation, and the magnitude of the estimate.

6.2 Evaluating the Frequency Estimation Process

A natural question at this point is how well the frequency estimates produced by our tools match the actual frequencies. To evaluate the accuracy of the estimates, we ran a suite of programs twice: once using the profiling tools, and once using dcpix, a pixie-like tool that instruments both basic blocks and edges at branch points to obtain execution counts. We then compared the estimated execution counts FP, where F is the frequency estimate and P the sampling period, to the measured execution counts – the values should be approximately equal (modulo sampling error) for programs whose execution is deterministic.

For this experiment, we used a subset of the SPEC95 suite. The subset contains the "base" versions of all floating point benchmarks, and the "peak" versions of all integer benchmarks except ijpeg. The other executables lacked the relocation symbols required by dcpix, and the instrumented version of ijpeg did not work. The profiles were generated by running each program on its SPEC95 workload three times.

Figure 8 is a histogram showing the results for instruction frequencies. The x-axis is a series of sample

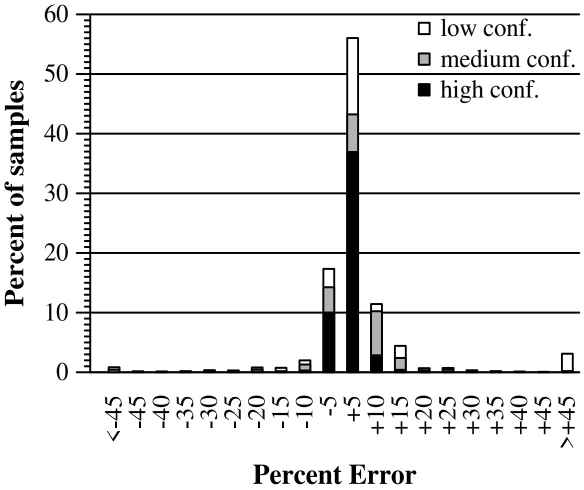

Figure 8: Distribution of Errors in Instruction Frequencies (Weighted by CYCLES Samples)

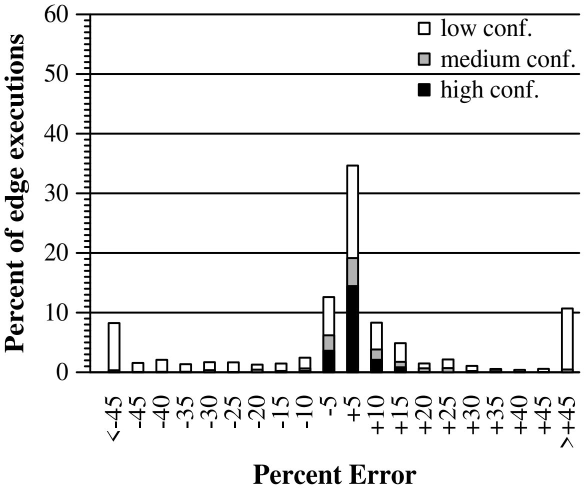

Figure 9: Distribution of Errors in Edge Frequencies (Weighted by Edge Executions)

buckets. Each bucket covers a range of errors in the estimate, *e.g.*, the -15% bucket contains the samples of instructions where FP was between $.85$ and $.90$ times the execution count. The y-axis is the percentage of all CYCLES samples.

As the figure shows, 73% of the samples have estimates that are within 5% of the actual execution counts; 87% of the samples are within 10%; 92% are within 15%. Furthermore, nearly all samples whose estimates are off by more than 15% are marked low confidence.

Figure 9 is a measure of the accuracy of the frequency estimates of edges. Edges never get samples, so here the y-axis is the percentage of all edge executions as measured by dcpix. As one might expect, the edge frequency estimates, which are made indirectly using flow constraints, are not as accurate as the block frequency estimates. Still, 58% of the edge executions have estimates within 10%.

To gauge how the accuracy of the estimates is affected by the number of CYCLES samples gathered, we compared the estimates obtained from a profile for a single run of the integer workloads with those obtained from 80 runs. For the integer workloads as a whole, results in the two cases are similar, although the estimates based on 80 runs are somewhat more tightly clustered near the -5% bucket. *E.g.,* for a single run, 54% of the samples have estimates within 5% of the actual execution counts; for 80 runs, this increases to 70%. However, for the individual programs such as gcc on which our analysis does less well using data from a small number of runs, the estimates based on 80 runs are significantly better. With a single run of the gcc workload, only 23% of the samples are within 5%; with 80 runs, this increases to 53%.

Even using data from 80 runs, however, the $>$45% bucket does not get much smaller for gcc: it decreases from 21% to 17%. We suspect that the samples in this bucket come from frequency equivalence classes with only one or two issue points where dynamic stalls occur regularly. In this case, gathering more CYCLES samples does not improve the analysis.

The analysis for estimating frequencies and identifying culprits is relatively quick. It takes approximately 3 minutes to analyze the suite of 17 programs, which total roughly 26 MB of executables. Roughly 20% of the time was spent blocked for I/O.

6.3 Identifying Culprits

Identifying which instructions stalled and for how long reveals *where* the performance bottlenecks are, but users (and, eventually, automatic optimizers) must also know *why* the stalls occurred in order to solve the problems. In this section, we outline the information our tools offer, how to compute it, and how accurate the analysis is.

Our tools provide information at two levels: instruction and procedure. At the instruction level, we annotate each stall with culprits (*i.e.*, possible explanations) and, if applicable, previous instructions that may have caused the stall. Culprits are displayed as labeled bubbles between instructions as previously shown in Figure 2. For example, the analysis may indicate that an instruction stalled because of a D-cache miss and point to the load instruction fetching the operand that the stalled instruction needs. At the procedure level, we summarize the cycles spent in the procedure, showing how many have gone to I-cache misses, how many to D-cache misses, etc., by aggregating instruction-level data. A sample summary is shown earlier in Figure 4. With these summaries, users can quickly identify and focus their effort on the more important performance issues in any given procedure.

For each stall, we list all possible reasons rather than a single culprit because reporting only one culprit would often be misleading. A stall shown on the analysis output is the average of numerous stalls that occurred during profiling. An instruction may stall for different reasons on different occasions or even for multiple reasons on the same occasion. For example, an instruction at the beginning of a basic block may stall for a branch misprediction at one time and an I-cache miss at another, while D-cache misses and write-buffer overflow may also contribute to the stall if that instruction stores a register previously loaded from memory.

To identify culprits for stalls, we make use of a variety of information. Specifically, we need only the binary executable and sample counts for CYCLES events. Sample counts for other types of events are taken into consideration if available, but they are optional. Source code is not required. Neither is symbol table information, although the availability of procedure names would make it easier for users to correlate the results with the source code.

Our analysis considers both static and dynamic causes of stalls. For static causes, we schedule instructions in each basic block using an accurate model of the processor issue logic and assuming no dynamic stalls. Detailed record-keeping provides how long each instruction stalls due to static constraints, why it stalls, and which previously issued instructions may cause it to stall. These explain the static stalls. Additional stall cycles observed in the profile data are treated as dynamic stalls.

To explain a dynamic stall at an instruction, we follow a "guilty until proven innocent" approach. Specifically, we start from a list of all possible reasons for dynamic stalls in general and try to rule out those that are impossible or extremely unlikely in the specific case in question. Even if a candidate cannot be eliminated, sometimes we can estimate an upper bound on how much it can contribute to the stall. When uncertain, we assume the candidate to be a culprit. In most cases, only one or two candidates remain after elimination. If all have been ruled out, the stall is marked as unexplained, which typically accounts for under 10% of the samples in any given procedure (8.6% overall in the entire SPEC95 suite). The candidates we currently consider are I-cache misses, D-cache misses, instruction and data TLB misses, branch mispredictions, write-buffer overflows, and competition for function units, including the integer multiplier and floating point divider. Each is ruled out by a different technique. We illustrate this for I-cache misses.

The key to ruling out I-cache misses is the observation that an instruction is extremely unlikely to stall due to an I-cache miss if it is in the same cache line as every instruction that can execute immediately before it[4]. More specifically, we examine the control flow graph and the addresses of instructions. If a stalled instruction is *not* at the head of a basic block, it can stall for an I-cache miss if and only if it lies at the beginning of a cache line. If it is at the head of a basic block, however, we can determine from the control flow graph which basic blocks may execute immediately before it. If their last instructions are all in the same cache line as the stalled instruction, an I-cache miss can be ruled out. For this analysis, we can ignore basic blocks and control flow edges executed much less frequently than the stalled instruction itself.

If IMISS event samples have been collected, we can use them to place an upper bound on how many stall cycles can be attributed to I-cache misses. Given the IMISS count on each instruction and the sampling period, we estimate how many I-cache misses occurred at any given instruction. From this estimate and the execution frequency of the instruction, we then compute the upper bound on stall cycles by assuming pessimistically that each I-cache miss incurred a cache fill all the way from memory.

How accurate is the analysis? Since in any nontrivial program there is often no way, short of detailed simulation, to ascertain why individual instructions stalled, we cannot validate our analysis directly by comparing its results with some "correct" answer. Instead, we evaluate it indirectly by comparing the number of stall cycles it attributes to a given cause with the corresponding sample count from event sampling, which serves

[4]Even so, an I-cache miss is still possible in some scenarios: the stalled instruction is executed immediately after an interrupt or software exception returns, or the preceding instruction loads data that happen to displace the cache line containing the stalled instruction from a unified cache. These scenarios are usually rare.

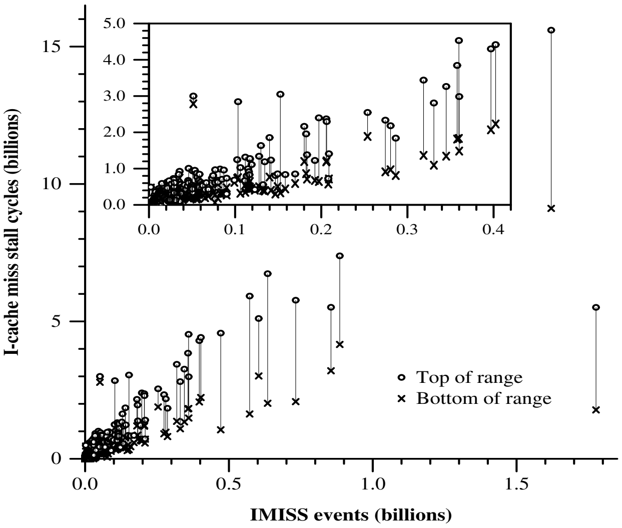

Figure 10: Correlation between numbers of I-cache miss stall cycles and of IMISS events for procedures in SPEC95 benchmark suite

as an alternative measure of the performance impact of the same cause. Though not a direct quantitative metric of accuracy, a strong correlation would suggest that we are usefully identifying culprits. (Since events can have vastly different costs, even exact event counts may not produce numbers of stall cycles accurate enough for a direct comparison. For example, an I-cache miss can cost from a few to a hundred cycles, depending on which level of the memory hierarchy actually has the instruction.) Again, we illustrate this validation approach with I-cache misses.

Figure 10 plots I-cache miss stall cycles against IMISS events for the procedures accounting for 99.9% of the execution time of each benchmark in the SPEC95 suite, with part of the main graph magnified for clarity. Each of the 1310 procedures corresponds to a vertical bar. The x-axis is the projected number of I-cache misses in that procedure, calculated by scaling the IMISS counts by the sampling period. The y-axis is the number of stall cycles attributed to I-cache misses by our tools, which report a range because some stall cycles may be caused only in part by I-cache misses[5].

Figure 10 shows that the stall cycles generally increase with the IMISS counts, with each set of endpoints clustering around a straight line except for a few outlier pairs. In more quantitative terms, the correlation coefficients between the IMISS count of each procedure and the top, bottom, and midpoint of the corresponding range of stall cycles are 0.91, 0.86, and 0.90 respectively, all suggesting a strong (linear) correlation. We would expect some points to deviate substantially from the majority because the cost of a cache miss can vary widely and our analysis is heuristic. For example, Figure 10 has two conspicuous outliers near (0.05,3) and (1.8,4). In the first case, the number of stall cycles is unusually large because of an overly pessimistic assumption concerning a single stall in the compress benchmark of SPECint95. In the second case, the number is smaller than expected because the procedure (`twldrv` in fpppp of SPECfp95) contains long basic blocks, which make instruction prefetching especially effective, thus reducing the penalty incurred by the relatively large number of cache misses.

[5]To isolate the effect of culprit analysis from that of frequency estimation in this experiment, the analysis used execution counts measured with instrumented executables as described in Section 6.2.

7 Future Directions

There are a number of interesting opportunities for future research. We plan to focus primarily on new profile-driven optimizations that can exploit the fine-grained information supplied by our analysis tools. Work is already underway to drive existing compile-time, link-time, and binary-rewriting optimizations using profile data, and to integrate optimizers and our profiling system into a single "continuous optimization" system that runs in the background improving the performance of key programs.

We also plan to further optimize and extend our existing infrastructure. We are currently investigating hardware and software mechanisms to capture more information with each sample, such as referenced memory addresses, register values, and branch directions. We have already prototyped two general software extensions: instruction interpretation and double sampling.

Interpretation involves decoding the instruction associated with the sampled PC, and determining if useful information should be extracted and recorded. For example, each conditional branch can be interpreted to determine whether or not the branch will be taken, yielding "edge samples" that should prove valuable for analysis and optimization. Double sampling is an alternate technique that can be used to obtain edge samples. During selected performance counter interrupts, a second interrupt is setup to occur immediately after returning from the first, providing two PC values along an execution path. Careful coding can ensure that the second PC is the very next one to be executed, directly providing edge samples; two or more samples could also be used to form longer execution path profiles.

We are also developing a graphical user interface to improve usability, as well as tools for interactively visualizing and exploring profile data. Finally, we are working with hardware designers to develop sampling support for the next generation of Alpha processors, which uses an out-of-order execution model that presents a number of challenges.

8 Conclusions

The DIGITAL Continuous Profiling Infrastructure transparently collects complete, detailed profiles of entire systems. Its low overhead (typically 1–3%) makes it practical for continuous profiling of production systems. A suite of powerful profile analysis tools reveals useful performance metrics at various levels of abstraction, and identifies the possible reasons for all processor stalls.

Our system demonstrates that it is possible to collect profile samples at a high rate and with low overhead. High-rate sampling reduces the amount of time a user must gather profiles before using analysis tools. This is especially important when using tools that require samples at the granularity of individual instructions rather than just basic blocks or procedures. Low overhead is important because it reduces the amount of time required to gather samples and improves the accuracy of the samples by minimizing the perturbation of the profiled code.

To collect data at a high rate and with low overhead, performance-counter interrupt handling was carefully designed to minimize cache misses and avoid costly synchronization. Each processor maintains a hash table that aggregates samples associated with the same PID, PC, and EVENT. Because of workload locality, this aggregation typically reduces the cost of storing and processing each sample by an order of magnitude. Samples are associated with executable images and stored in on-disk profiles.

To describe performance at the instruction-level, our analysis tools introduce novel algorithms to address two issues: how long each instruction stalls, and the reasons for each stall. To determine stall latencies, an average CPI is computed for each instruction, using estimated execution frequencies. Accurate frequency estimates are recovered from profile data by a set of heuristics that use a detailed model of the processor pipeline and the constraints imposed by program control-flow graphs to correlate sample counts for different instructions. The processor-pipeline model explains static stalls; dynamic stalls are explained using a "guilty until proven innocent" approach that reports each possible cause not eliminated through careful analysis.

Our profiling system is freely available via the Web [7]. Dozens of users have already successfully used our system to optimize a wide range of production software, including databases, compilers, graphics accelerators, and operating systems. In many cases, detailed instruction-level information was essential for pinpointing and fixing performance problems, and continuous profiling over long periods was necessary for obtaining a representative profile.

Acknowledgements

We would like to thank Mike Burrows, Allan Heydon, Hal Murray, Sharon Perl, and Sharon Smith for helpful comments that greatly improved the content and presentation of this paper; the anonymous referees for SOSP and TOCS also provided numerous helpful comments. We would also like to thank Dawson Engler for initially suggesting the use of inter-processor interrupts to avoid expensive synchronization operations in the interrupt handler, Mitch Lichtenberg for his work on the Alpha/NT version of our system and in general for his help and suggestions on the project, and the developers of iprobe for supplying us with source code that helped us get off the ground in building the early versions of our data collection system. Finally, we would like to thank Gary Carleton and Bob Davies of Intel for answering our questions about VTune and Marty Itzkowitz of SGI for answering our questions about SpeedShop.

References

[1] T. E. Anderson and E. D. Lazowska. Quartz: A tool for tuning parallel program performance. In *Proceedings of the ACM SIGMETRICS 1990 Conference on Measurement & Modeling of Computer Systems*, pages 115–125, Boulder, Colorado, May 1990.

[2] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[3] D. Blickstein *et al.* The GEM optimizing compiler system. *Digital Technical Journal*, 4(4), 1992.

[4] D. Carta. Two fast implementations of the 'minimal standard' random number generator. *Communications of the Association for Computing Machinery*, 33(1):87–88, January 1990.

[5] R. Cohn, D. Goodwin, P. G. Lowney, and N. Rubin. Spike: An optimizer for Alpha/NT executables. In *USENIX Windows NT Workshop*, Seattle, Washington, Aug 1997.

[6] R. Cohn and P. G. Lowney. Hot cold optimization of large Windows/NT applications. In *29th Annual International Symposium on Microarchitecture (Micro-29)*, Paris, France, December 1996.

[7] DIGITAL Continuous Profiling Infrastructure project. http://www.research.digital.com/SRC/dcpi/.

[8] Digital Equipment Corporation. *Alpha 21164 Microprocessor Hardware Reference Manual*. Maynard, MA, 1995. Order Number EC-QAEQB-TE.

[9] Digital Equipment Corporation. *DECchip 21064 and DECchip 21064A Alpha AXP Microprocessors Hardware Reference Manual*. Maynard, MA, 1995. Order Number EC-Q9ZUA-TE.

[10] Aaron J. Goldberg and John L. Hennessy. MTOOL: An Integrated System for Performance Debugging Shared Memory Multiprocessor Applications. *IEEE Trans. on Parallel and Distributed Systems*, pages 28–40, January 1993.

[11] S. Graham, P. Kessler, and M. McKusick. `gprof`: A call graph execution profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982.

[12] M. Hall *et al.* Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[13] Iprobe. Digital internal tool.

[14] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 171–185, Orlando, Florida, 1994.

[15] J. D. McCalpin. Memory bandwidth and machine balance in high performance computers. *IEEE Technical Committee on Computer Architecture Newsletter*, December 1995. http://www.cs.virginia.edu/stream.

[16] J. McCormack, P. Karlton, S. Angebranndt, and C. Kent. x11perf. http://www.specbench.org/gpc/xpc.static/index.html.

[17] MIPS Computer Systems. *UMIPS-V Reference Manual (pixie and pixstats)*. Sunnyvale, CA, 1990.

[18] prof. Digital Unix `man` page.

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

[20] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3(3), Fall 1995.

[21] R. Sites and R. Witek. *Alpha AXP Architecture Reference Manual*. Digital Press, Newton, MA, 1995.

[22] The Standard Performance Evaluation Corporation. http://www.specbench.org/osg/spec95.

[23] Transaction Processing Performance Council. http://www.tpc.org/bench.descrip.html.

[24] Vtune: Intel's visual tuning environment. http://developer.intel.com/design/perftool/vtune.

[25] M. Zagha *et al.* Performance analysis using the MIPS R10000 performance counters. In *Proceedings of Supercomputing*, Pittsburgh, Pennsylvania, November 1996.

[26] X. Zhang *et al.* Operating system support for automated profiling & optimization. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, St. Malo, France, Oct 1997.