2004-11-16

Does the Halting Necessary for Hardware Trace Collection Inordinately Perturb the Results?

Myles G. Watson
Brigham Young University - Provo

DOES THE HALTING NECESSARY FOR HARDWARE TRACE COLLECTION INORDINATELY PERTURB THE RESULTS?

by

Myles G. Watson

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

December 2004

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Myles G. Watson

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	J. Kelly Flanagan, Chair
Date	Michael A. Goodrich
Date	Charles D. Knutson

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Myles G. Watson in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	J. Kelly Flanagan Chair, Graduate Committee

Accepted for the Department

David W. Embley
Graduate Coordinator

Accepted for the College

G. Rex Bryce, Associate Dean
College of Physical and Mathematical Sciences

ABSTRACT

DOES THE HALTING NECESSARY FOR HARDWARE TRACE COLLECTION INORDINATELY PERTURB THE RESULTS?

Myles G. Watson

Department of Computer Science

Master of Science

Processor address traces are invaluable for characterizing workloads and testing proposed memory hierarchies. Long traces are needed to exercise modern cache designs and produce meaningful results, but are difficult to collect with hardware monitors because microprocessors access memory too frequently for disks or other large storage to keep up. The small, fast buffers of the monitors fill quickly; in order to obtain long contiguous traces, the processor must be stopped while the buffer is emptied. This halting may perturb the traces collected, but this cannot be measured directly, since long uninterrupted traces cannot be collected. We make the case that hardware performance counters, which collect runtime statistics without influencing execution, can be used to measure halting effects. We use the performance counters of the Pentium 4 processor to collect statistics while halting the processor as if traces

were being collected. We then compare these results to the statistics obtained from unhalted runs. We present our results in terms of which counters are affected, why, and what this means for trace-collection systems.

ACKNOWLEDGMENTS

Many thanks to Kelly Flanagan for his patience, support, and financial assistance. Thanks also to my wife, Christie, for cheering me on.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This thesis uses the hardware performance counters available in the Pentium 4 processor to explore the effects of halting the processor for trace collection. Trace collection allows a designer to use memory access histories from current processors to test proposed memory hierarchies.

This chapter introduces the problem to be solved and lays out the organization for the rest of the thesis. It starts by explaining the growing difference between processor and DRAM speeds, and how memory hierarchies help to mitigate the effects of this difference. It then introduces trace-driven simulation as an important tool for evaluating memory hierarchy designs. After focusing on hardware trace collection using BYU Address Collection Hardware (BACH)[1, 2], we point out some possible sources of perturbation introduced by the system. This perturbation is quantified by the experiments in this work, using hadware performance counters.

1.1 Processor Memory Speed Gap

The widening gap between CPU and main memory speeds makes memory subsystem design critical. According to Hennessey and Patterson [3], CPU speeds have been growing at 55% per year, while main memory (DRAM) speeds have been growing at 7% per year. Figure 1.1 illustrates the effect of this difference (note the logarithmic

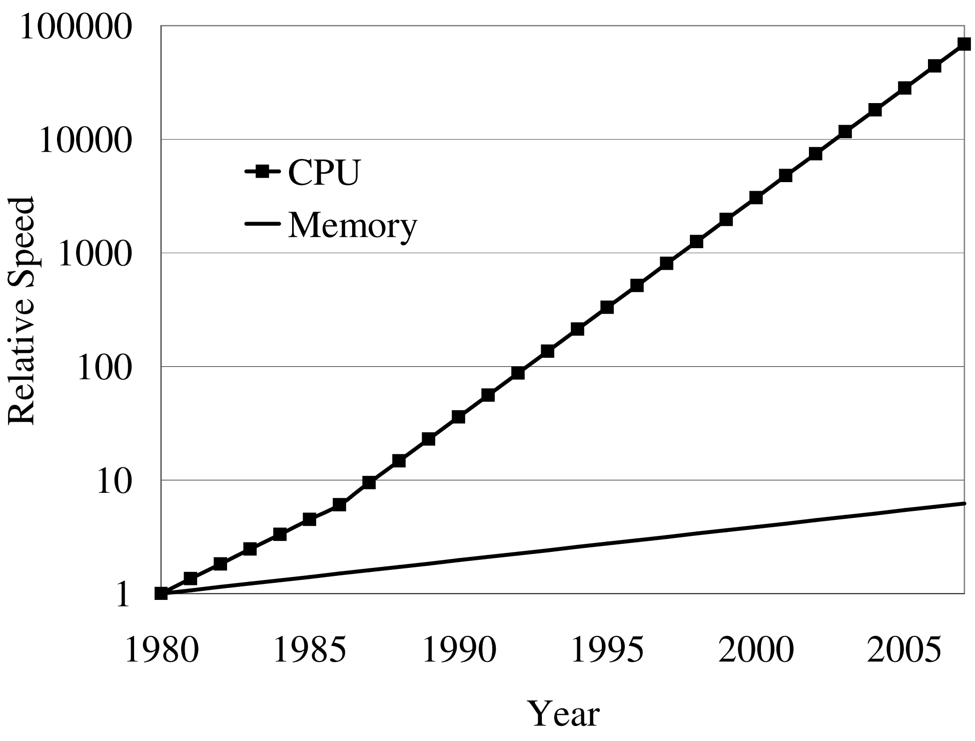

Figure 1.1: The CPU-DRAM performance gap. Notice the logarithmic scale of the Y axis.

---

scale of the Y axis). The difference between the rates can be attributed to the difference in design goals. While processor designers are focused on increasing speed, DRAM suppliers are focused on increasing densities and therefore capacity (DRAM sizes are doubling every two years [4]). In order to continue to increase in performance, CPUs must access main memory less frequently. One way to do this is by using a hierarchical memory design.

Memory Hierarchies

The role of memory hierarchies is to approach the speed of the processor while managing overall system cost. Caches do this by organizing small fast blocks of memory (usually SRAM) to take advantage of temporal and spatial locality [5]. Expensive SRAM can run at the speed of the processor, but the cost is prohibitive for main memory in current systems. If the most frequently used data is kept in the cache, then for much of the time the processor will be able to execute as though main memory were as fast as the cache.

There are many parameters to a memory hierarchy design. These include the number of levels in the hierarchy and bandwidth between levels. For each level, the capacity, associativity, block size, and replacement policy must be chosen. Since the optimal values are workload dependent and most of these parameters contribute substantially to the cost, the trade-offs must be weighed carefully.

Study Methods

Researchers employ several methods to study the trade-offs among cache parameters, price, and performance. These methods include queuing theory, analytical models, and simulation. Because of the difficulty in accurately modeling the interactions of components of the system, simulation is often the method of choice. Trace-driven simulation uses the memory access history of a workload as input to a simulator in order to obtain cache and performance measurements.

1.2 Trace-Driven Simulation

Trace-driven memory simulation has been an important tool in the evaluation of memory hierarchies for many years. There are several types of trace-driven simulation, differentiated by how traces are collected, stored, and processed [6]. Traces can be collected using functional simulation of the architecture [7], instrumenting the source or executable code of a benchmark [8], or connecting a hardware probe to the processor pins. The traces can then be stored in a raw format like BYU Address Trace Format [9] and Dinero [10], in a compressed format like PDATS [11, 12], or fed directly to a simulator without intermediate storage [7, 13, 14].

As memory hierarchies continue to grow in size, the length of traces needed to exercise them also grows. This growth makes all phases of trace-driven simulation more difficult, and influences implementation decisions for tracing systems.

1.3 Trace Collection

In this work we are more concerned with trace collection than trace storage or trace processing, so we explain the relative merits of trace collection using simulation, code instrumentation, or hardware probes. These three methods are part of a continuum; simulation introduces the most slowdown and is the most flexible, and hardware probes are the least flexible and introduce the least slowdown.

Simulation

Simulation is the most flexible approach because it allows trace collection for various hardware with the same system. It allows researchers to study hardware that is not available, and the hardware or software contributions to total performance. One drawback is that it slows execution by several orders of magnitude based on the level of detail at which it is run, and is therefore prone to errors due to insufficient detail. Other drawbacks are the time needed to implement an accurate simulator of a modern architecture and the lack of details necessary to do so.

Instrumentation

Instrumenting the benchmark is done by inserting code snippets, which record information available during execution. This information may include which basic block is being executed or what virtual memory addresses are being accessed. It allows the program to run at much closer to normal execution speeds. Adding instrumentation changes the memory footprint and the execution path of a benchmark. These changes introduce slowdowns, which change the speed of other system components relative to the CPU. Another drawback is that it is architecture specific, requiring the researcher to have the hardware. It also requires an instrumented operating system to collect complete traces.

Hardware Probes

Using hardware probes is the most machine specific of these techniques. A hardware probe is designed to connect electrically to the processor pins and record the memory transactions. It allows the collection of complete address traces [1]. Here complete means that both operating system and user references are recorded. Tracing with a hardware probe allows the processor to run at full speed until the fast memory, or buffer, fills. This is one drawback, because the buffers are not large enough to hold interesting traces; the processor must be halted while the buffer is emptied in order to collect contiguous traces longer than one buffer in length. Another drawback is that probes are expensive to build or buy. This is especially true considering the rate of replacement necessary to stay abreast of current processor technology.

Figure 1.3 represents the halting of a workload for trace collection. The goal is to construct contiguous traces of halted runs that are indistinguishable from a trace captured without halting the processor. Since an interesting trace of the memory references is larger than a practical buffer, it must be collected in pieces and reassembled. In the figure this means that the results of a cache simulation using the four buffers of scenario B or the seven buffers of scenarios C and D would be equivalent to the cache simulation using the single trace collected in scenario A. Scenario A shows the normal execution of the program. Scenarios B, C, and D represent the same program while traces are being collected by different tracing systems. The tracing system in scenario B has twice as large of a buffer as C or D, and B and C empty their buffers twice as fast as D. Since the cost of a tracing system increases with the speed and depth of the buffers, it is of interest to know what effect these parameters have on the system being traced, and therefore on the workload being collected.

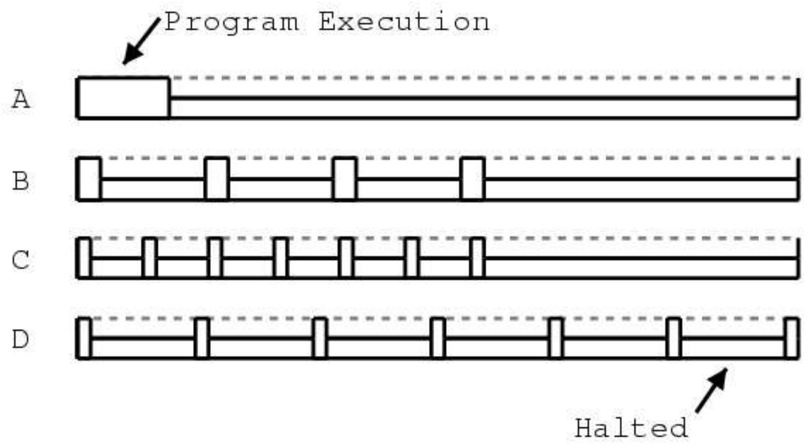

Figure 1.2: The general method of hardware trace collection with a finite buffer. Program execution refers to the benchmark running, when the trace is being collected. Note that the time between interruptions increases with larger buffers, and the time the processor spends halted decreases as the speed of emptying the buffer increases.

1.4 BACH: BYU Address Collection Hardware

BACH collects complete address traces using a hardware probe [1]. The current implementation of BACH uses Tektronix microprocessor probes and logic analyzers that can buffer 8 million memory references. When the buffer fills, the logic analyzer causes a high priority interrupt on the system under test, which spins in a tight loop until the logic analyzer empties its buffer and releases the interrupt. The number of times this interrupt service routine is entered depends on the length of the buffer and the length of the trace to be collected. One BACH trace of the SPEC CPU2000 benchmark 181.mcf with caches enabled is over 6.4 billion references long, meaning that more than 800 buffers must be collected [9].

Possible Sources of Error

BACH introduces some perturbation in traces that it collects. Some possible sources of error are memory references, bus traffic, and changes in processor state due to the tracing device driver; another is the reordering of interrupt handlers.

Each time the device driver on the system under test enables tracing, a few memory references from the driver code will be captured. This effect can be minimized or eliminated with careful post-processing of the trace to remove references in the memory range of the driver. The changes in the machine state caused by the tracing device driver can not be removed. Cache lines, page table entries, and other resources that are used by the device driver displace state used by the workload and change subsequent execution. This is minimized by writing driver code that uses as few resources as possible.

Due to the asynchronous nature of interrupts, they continue to occur while the processor is waiting for the logic analyzer buffer to empty. Interrupts from sources such as the timer, disk drive, and network card queue up and wait until tracing is re-enabled. Since a buffer takes longer than a timer tick to empty, the beginning of every buffer will contain at least the references for the timer tick service routine. In order to minimize the number of interrupts that queue up, BACH traces are collected with the network disabled and in single-user mode in Linux.

1.5 Summary

In this chapter, we motivated the use of trace-driven simulation to evaluate memory hierarchies. We then discussed trace collection, and the use of hardware probes. We focused on BACH, and pointed out potential sources of inaccuracies.

1.6 Thesis Statement and Layout

We submit that halting the processor is a negligible source of error in trace-driven simulation with a sufficiently large tracing buffer. In order to make that case, we present experiments and results that measure the differences in execution caused by halting the processor with varying frequencies and for various lengths of time. Chapter 2 describes the experiments we designed and details hardware performance counters. Chapter 3 presents our results, and Chapter 4 gives our conclusions and possible future work.

Chapter 2

Experiments

This chapter starts by explaining what we wanted to accomplish and the experiments that we designed. It then describes hardware performance counters and gives some details about the counters available on the Pentium 4 processor. Finally it explains how we chose the counters for our experiments, how we access them, and the statistical methods we use.

2.1 Using Performance Counts to Detect Perturbation

We base our experiments on the idea that performance counts can be used to characterize periods of execution. In other words, a benchmark run multiple times should produce similar performance counts for each run. Perturbation, or interference, from an outside source should affect the performance counts to the extent that it affects the execution of the benchmark. In this work we wish to quantify the perturbation introduced by halting the execution of a benchmark for trace collection.

Since we are interested in quantifying the perturbation introduced by BACH, the halting method is fixed in our experiments, and we vary the time for which the processor is halted and the frequency of halting. These parameters correspond to the speed with which the tracing buffer can empty and the depth of the buffer, respectively.

Varying Halting Times

In order to quantify the effect of increasing halting times, we chose two frequencies at which to halt the processor. For the first, we chose the minimum amount of time for BACH's buffers to fill. The second we chose to be four times faster to give us another data point representing the same logic analyzer collecting references from a faster bus.

We calculated the minimum time to fill BACH's buffer starting with the maximum frequency with which a 2.4 GHz Pentium 4 processor with a 400 MHz front-side bus can make memory requests. Since the 400 MHz bus makes requests at 100MHz, and there are at least six bus cycles per request, there can be at most 16 million requests per second. Our logic analyzer can buffer 8 million requests, or one half second of requests at the maximum rate. We thus chose to halt the processor every 1.2 billion cycles, or approximately every one half second.

We compare not halting the processor, halting it for the shortest amount of time possible, and spinning in a tight loop for 1/4, 1/2, 1, 2, or 8 seconds. Each time the processor is halted, we disable interrupts and spin in a tight loop for a given halting time. We approximate the halting time with a nested loop where the inner loop executes 1.2 Million times, or about a millisecond. This means that total execution time for a benchmark is given by equation 2.1, where t is the number of seconds the benchmark is halted each time, x is the normal time to completion for the benchmark, and f is the execution time in seconds between halting times. Halting every 1/8 second for 2 seconds therefore increases the runtime to $17x$. In order to reduce total execution time, when halting every 1/8 second we consider fewer wait times, see Table 2.1.

$$tx/f + x \tag{2.1}$$

Execution Time Between Halting	Halting Time
No Halting	N/A
1/8 second	0, 1/2, and 2 seconds
1/2 second	0, 1/4, 1/2, 1, 2, and 8 seconds

Table 2.1: Experiments for Determining the Effect of Halting for Different Lengths of Time. We consider fewer hating times when halting every 1/8 second to reduce run times.

Varying Halt Frequencies

In order to quantify the effect of halting frequency, we held the halting time constant and varied the frequency of halting. These times were not chosen to represent the current halting time of BACH, which is on the order of tens of minutes. Instead we chose times that we feel represent feasible solutions, given the sustained write bandwidth of modern disks and arrays. We chose halting times of every two seconds and every one half second.

We compare not halting the processor to halting the processor every 1/10, 1/2, 1, 2, and 8 seconds. Again to save time we use fewer data points when halting for two seconds. Table 2.2 shows the runs for these experiments.

Halting Time	Execution Time Between Halting
No Halting	N/A
1/2 second	1/10, 1/2, 1, 2, 8 seconds
2 seconds	1/2 and 2 seconds

Table 2.2: Experiments for Determining the Effect of Halting With Different Frequencies. Note that the number of runs for this experiment was reduced in the same manner as the previous experiment to reduce run times.

2.2 Hardware Performance Counters

Background

Modern microprocessors are complex. Their decoupled out-of-order and speculative execution engines make it difficult to predict how well specific code will run [15]. In order to help find and fix performance problems, architects include performance counters. These counters allow the counting of architectural events, such as cache misses and branch mispredictions, without significant overhead in terms of execution time, and without modifying the code. This allows designers to improve the architecture and developers to tune their algorithms to the hardware.

Designers can run code segments and compare the expected and actual counts. If there are significant differences, they narrow the search for the reason the code behaves poorly. The code can then be rewritten to perform better, and/or the architecture can be modified in the next revision. There are several software tools that allow the programmer to access performance counts [16, 17, 18, 19].

2.3 Performance Monitoring Features of the Pentium 4 Processor

The Intel Pentium 4 processor has more counters available for simultaneous event counting than its predecessors [20, 21]. In the Pentium Pro, Pentium II, and Pentium III processors, there are only two counters available for simultaneous use. This limits the experiments that can be run because of the difficulties involved with correlating the counts among different runs. The Pentium 4 processor has 18 counters available, which can each be configured to count a variety of events.

Organization and Setup

The performance counters on the Pentium 4 processor are distributed in groups throughout the chip to reduce routing overhead [16]. Groups consist of counters, event selection control registers (ESCRs), and counter configuration control registers (CCCRs), as shown in Figure 2.1. Each counter has its own CCCR, which selects

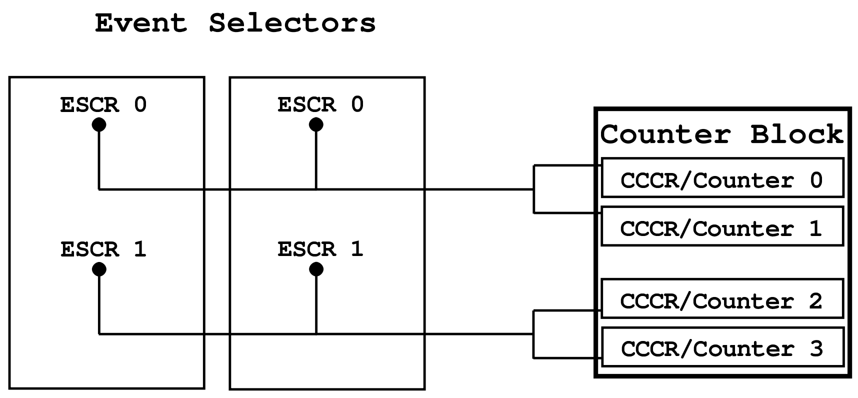

Figure 2.1: Performance Counter Blocks in the Pentium 4 processor. Note that each event selector may be used to count various events. For simplicity we have only shown two event selectors with this counter block.

---

an ESCR's signals to count and sets filtering options and overflow conditions. Some events may happen more than once per clock cycle, so the counters can increment by up to fifteen each clock cycle. The CCCRs' filtering mechanisms allow the programmer to specify event counting thresholds, count inversion, and edge filtering.

Table 2.3 shows a few of the events that can be counted in the Pentium 4 processor, which block they belong to, and includes a brief description of the event. For example, to count retired mispredicted branches, either TBPU_ESCR0 or TBPU_ESCR1 is programmed to select event 0x05 with bits set corresponding to which types of branches should be counted. The CCCR connected to the chosen ESCR is set to select that ESCR's increment signals, and the corresponding filter and overflow bits are set appropriately.

Event Based Sampling

Event based sampling (EBS) is another profiling method that is supported by the Pentium 4 processor. In event based sampling, an interrupt is generated when a counter overflows. The interrupt handler can then record some subset of the state

Counter Name	Block	Description
Retired_mispred_branch_type	TBPU	Retiring mispredicted branches by type.
Instr_retired	CRU	Retiring instructions which may be tagged or speculative.
Global_power_events	FSB	Time during which the processor is not halted.

Table 2.3: These are example Pentium 4 performance counters and their descriptions [21].

---

of the microprocessor, for example, the value of the program counter or architectural registers. In this way, a histogram of instructions that cause a specific event can be created. Using an interrupt to trigger the sample is referred to as imprecise EBS (IEBS). In IEBS, the overhead involved with triggering an interrupt and the large number of instructions that may currently be in flight can lead to the data being collected 50 instructions later than the one that caused the event. This can make fixing performance problems very difficult. As processors execute more and more aggressively, this will continue to get worse.

Precise Event Based Sampling

To help remedy this problem, the Pentium 4 processor has included precise EBS (PEBS). PEBS uses a micro-assist instead of an interrupt to handle the counter overflow. This means that the event is handled completely in hardware. The sample data gets stored to a buffer area that was previously allocated in memory, the counter gets reset, and the processor is ready for the next occurrence. With PEBS, nearly all samples will correspond to the instruction that caused the event.

Our Usage of EBS

We use IEBS to interrupt the processor after some number of clock cycles have passed. In the interrupt service routine that we wrote, no information is collected. Instead, the processor spins in a tight loop to simulate the time needed to empty the tracing buffer. If we had been able to use PEBS, it would have reduced the overhead involved, but it did not allow us to halt the machine. We could have collected the counter values each time the processor was halted in this way, but we wished to minimize overhead in our experiments.

2.4 Counter Selection

Now we address which counters we chose for our experiments. There are many events that can be counted on the Pentium 4 processor, up to 18 at once. Because of the variability in counts between runs, we decided to limit the number of counters so that we could collect them all in a single run.

Starting Point

We started with the counters used by Gomez et al. [22]. In their work they use the SPEC CPU2000 benchmarks to compare benchmark reduction methods. They argue that reduction methods are of little use if they significantly change the character of the benchmark. They compare dataset reduction and fast-forwarding techniques to the complete benchmark. The criteria for accuracy is the amount that the performance counts change. They use CPI (cycles per instruction), first and second level cache miss rates, branch misprediction rate, instruction and data TLB miss rates, and degree of speculation (percent instructions committed).

To their list, we added global power events. This counter is the preferred method for counting clock cycles, according to the Pentium 4 processor manual. We configure it to count all cycles when the counter is active. We use this cycle count with EBS

```
 1
 2        for (y=0;y<10000000;y++)
 3        {
 4            for (a=0;a<MISS_TIMES;a++)
 5                x += buffer[a*0x2000];   //Miss in the Cache
 6            a--;
 7            for (z=0;z<64-MISS_TIMES;z++)
 8                x += buffer[a*0x2000];   //Hit in the Cache
 9        }
10
```

Figure 2.2: This is an excerpt from SimpleMisses.c, a small program which behaves differently in the data cache depending on the value of MISS_TIMES.

to halt the processor based on the amount of time that it has been executing the benchmark.

Misses Benchmark

In order to become familiar with the performance counters and tools, we wrote some simple programs. One program was written to cause cache misses. It consists of nested for loops, which march through memory accessing locations that map to the same line, 32k apart. Figure 2.2 contains an excerpt of the code. Note that the number of memory loads and instructions executed remain the same for any setting of MISS_TIMES, because lines 5 and 8 are executed a total of 64 times.

First and Second Level Misses

Figure 2.3 is a scatter plot with the MISS_TIMES on the X axis, and the number of misses on the Y axis. When the associativity of the L1 cache is exceeded, the number of misses increases linearly with MISS_TIMES. We used the same program to test L2 miss counts, but the counts were not correlated with MISS_TIMES. There are two different ways that the Intel manual [21] suggests to count L2 misses, one of which is stated to be incorrect due to an erratum. The other only works when prefetching is disabled, which affects the execution of the benchmarks; we wanted to avoid this.

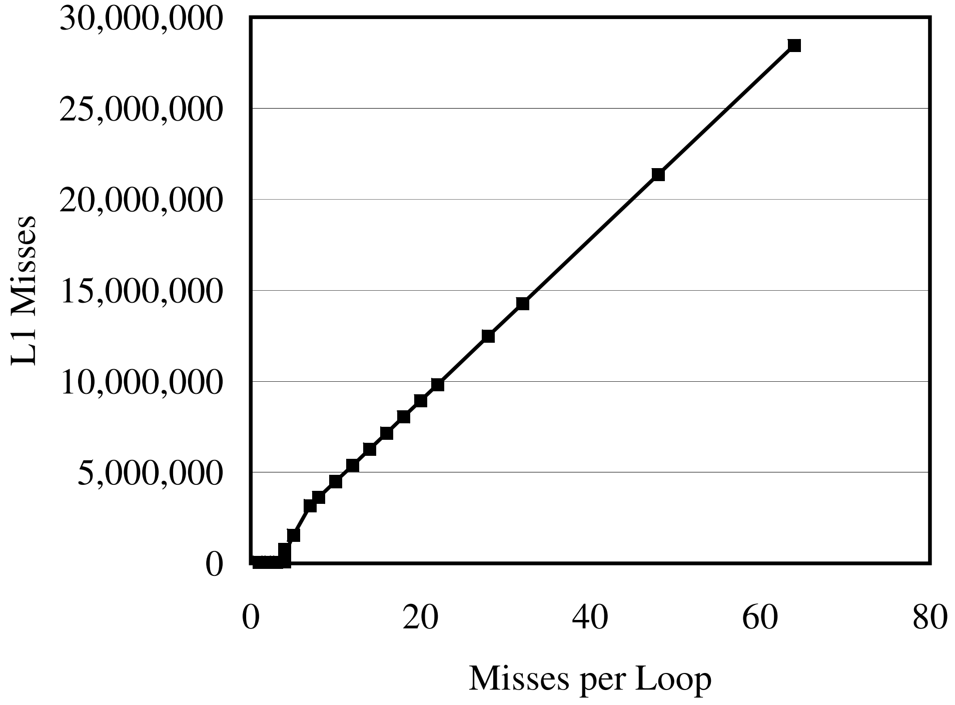

Figure 2.3: First level data cache misses from SimpleMisses.c for several values of MISS_TIMES. Notice that when the associativity of the first level cache is exceeded, the number of misses increases linearly with the value of MISS_TIMES

Bus Transactions

In order to get a count that was related to the L2 miss counts, we also configured the counters to count bus transactions and bus queue allocations. These metrics also yielded no meaningful correlations and were several orders of magnitude smaller than expected, so we abandoned the effort to measure L2 misses and bus activity.

64k Aliasing

When trying to find L2 correlations, we noticed that elapsed time did not always correspond as nicely as L1 misses to MISS_TIMES. We looked at other counters and found that 64k aliases correlated with elapsed time. 64k aliasing occurs when the fast match logic detects that the lower 16 bits of the address cause a hit in the cache. The processor then assumes that it was a hit and begins executing the instructions with the value from that location. When it is determined that the match was the result of aliasing, the instructions that depended on that value must be canceled and restarted. Our code is designed to miss the cache by accessing once every 32k, which increases our chance of 64k aliasing. Page allocation may be different from run to run, and therefore 64k aliasing also varies. We decided to include it in the counters we use.

Allocate Remaining Available Counters

From this point, we allocated metrics to the remaining counters. We added split loads, split stores, memory stalls due to the reorder buffer being full, and trace cache mode counters. Split loads and stores occur when there was a replay caused at the respective port. The reorder buffer is a structure that makes sure that instructions retire in the correct order. The trace cache has two modes, build and deliver. The trace cache is in build mode when it is decoding instructions. There are not enough documented events to use all the counters in the IQ group, so we end up using 16 performance counters and the time-stamp counter; see Table 2.4.

Selected Pentium 4 Processor Performance Counters
64k Aliases
Global Power Events
Conditional Branches
Conditional Branch Mispredictions
Data TLB Misses
Instruction TLB Reads and Writes
Instruction TLB Misses
Instructions Retired (All)
Instructions Retired (Non-Bogus)
Level 1 Cache Misses
Loads (Requires Two Counters)
Split Loads
Split Stores
Trace Cache Build Mode
Trace Cache Deliver Mode
Time Stamp Counter

Table 2.4: These are the final counters that were selected to measure the perturbation of halting. They were selected because they measure important performance events, and/or could be collected simultaneously.

2.5 Statistical Methods

We use statistical methods from the performance analysis book by Raj Jain [23]. In particular we make use of confidence intervals around the mean and an equation to determine necessary sample sizes. We use confidence intervals to make claims about statistically significant differences between halting frequencies and halt times. We use the equation to calculate sample sizes to find the number of runs necessary to limit the size of the confidence intervals with respect to the total counts.

Comparing Means Using Confidence Intervals

In order to determine the amount of perturbation introduced by halting the system, we use confidence intervals around the mean value of each counter to find statistically significant differences. The confidence interval allows us to place bounds on the value of the true mean with a given confidence. In other words, a 95% confidence interval of (23,26) means that there is a 95% certainty that the mean of the population is between 23 and 26. When comparing two alternatives, if the confidence intervals do not overlap, one can state with a given confidence that the mean of the two alternatives is different. If the confidence intervals do overlap, you cannot tell that the means are different with that level of confidence.

$$(\overline{x} - s z_{1-\alpha/2}/\sqrt{n}\ , \quad \overline{x} + s z_{1-\alpha/2}/\sqrt{n}\ ) \tag{2.2}$$

Confidence intervals are computed using the sample mean, standard deviation, number of samples, and the t distribution for a given confidence level and sample number. Equation 2.2 gives the formula for confidence intervals. The width of a confidence interval grows with increasing confidence (decreasing α), increasing standard deviation s, and shrinks with increasing sample size n. We decided to use 95% confidence intervals before running the experiments.

Determining Sample Size

Formula 2.3 yields the number of samples necessary to achieve a confidence interval of a certain size given a few samples from the distribution. Here z is the quantile of the unit normal distribution, s is the sample standard deviation, $\overline{x}$ is the sample mean, and r is the percent accuracy required. We used this formula to determine how many measurements were necessary to keep the variability to 5% of the counter values for unperturbed runs. For the integer SPEC CPU2000 benchmarks, we calculated that we needed 3 or fewer runs for all of the counters we were interested in. We decided to use 5 runs for each of our experiments in case there was increased variability in the counter values when the system was halted.

$$n = (100zs/r\overline{x}) \tag{2.3}$$

2.6 Experimental Environment and Details

In this section we explain the details of our implementation of the experiments. We describe the hardware and operating system, then the software, and then some implementation-specific details.

Hardware and Operating System

Our Experiments are run on a Dell Optiplex GX260 with a 2.4 GHz Pentium 4 processor and 512 MB of RAM. We have installed Redhat 9 with a modified Linux kernel based on 2.4.20-8 in order to access the performance counters. For each of our experiments, we run all of the integer SPEC CPU2000 benchmarks, and three of the floating-point benchmarks. The integer benchmarks were compiled with gcc, and the floating-point benchmarks were compiled with Fujitsu's fortran compiler. We run our experiments in single user mode in order to limit the number of active interrupts and mirror BACH's style of trace collection.

Software

The Brink and Abyss tools [24] provide an experimental environment in which to work. They consist of a Perl script (Brink), a C program that interfaces to the device driver (Abyss front end), and the device driver that accesses the counters (Abyss device driver).

Brink parses two files, a processor description file and an experiment configuration file. From these files it creates an input file for the Abyss front end that contains the path to the benchmark, initial counter values, sampling directives, and EBS directives. It then runs the Abyss front end and parses the output, writing log files, event counts, and archiving the input file.

The Abyss front end uses the directives from the input file to run the benchmark. It decides which device driver calls to issue with what frequencies, and formats the driver's output.

The Abyss device driver provides functions for reading and writing the performance monitoring counters and registers. It also includes routines for handling IEBS, PEBS, and the interrupt service routine (ISR) for the performance counter interrupt. Since the device driver needs specific interrupt support compiled into the kernel, it comes with the necessary patches.

Implementation Specific Details

To isolate the effects of halting the processor, we subtract obvious effects such as instructions executed in the wait loop, extra branches taken, etc. We multiply the number of times that the benchmark is halted by the number of extra events caused by each loop and by the number of times the loop is executed. For example, for each inner loop there is an extra branch and every millisecond there is another branch from the outer loop. This means that we subtract $th(1.2M + 1)$ branches from the total number, where t is the number of milliseconds the benchmark is halted each time and h is the number of times the benchmark is halted. The confidence intervals are small enough to make the extra th branches from the outside loop significant.

2.7 Summary

We started this chapter by outlining our experiments. We then described hardware performance counters and the counters available on the Pentium 4 processor, followed by which counters we chose for our experiments. We then explained the statistical methods we used, and some additional details.

Chapter 3

Results

In this chapter we discuss our experimental results. We start by describing and presenting graphical examples of statistically significant trends (as opposed to results that show no significant trend). We then address each experiment's results, describing the trends that were encountered, and their meanings. We show that although hardware trace collection perturbs the system very little, our methodology allows us to quantify the perturbation and make a recommendation for minimum buffer size. Throughout this chapter we urge the reader to pay attention to the scales of the graphs, as they were chosen to make the trends clear, not the magnitude of the counts.

3.1 Significant Trends

In order to characterize the effects of an experiment, we graphed the confidence intervals for each counter in each experiment. We then selected those graphs that showed statistically significant patterns, and tabulated the results. Figure 3.1 is an example where there is no statistically significant effect of increasing the halting time, and is included for contrast with figures 3.2, 3.3, and 3.4, which show examples of statistically significant effects.

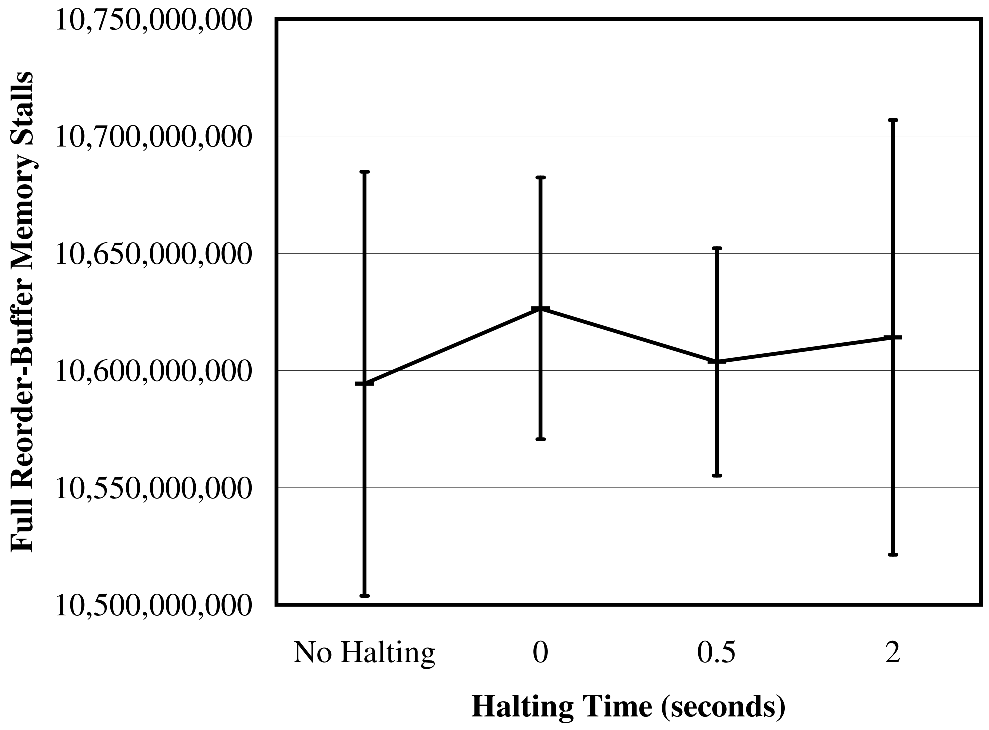

Figure 3.1: No statistically significant trend. This figure contains no trend, because the confidence intervals contain each other's means. Therefore, even though there is variation, it does not indicate a pattern.

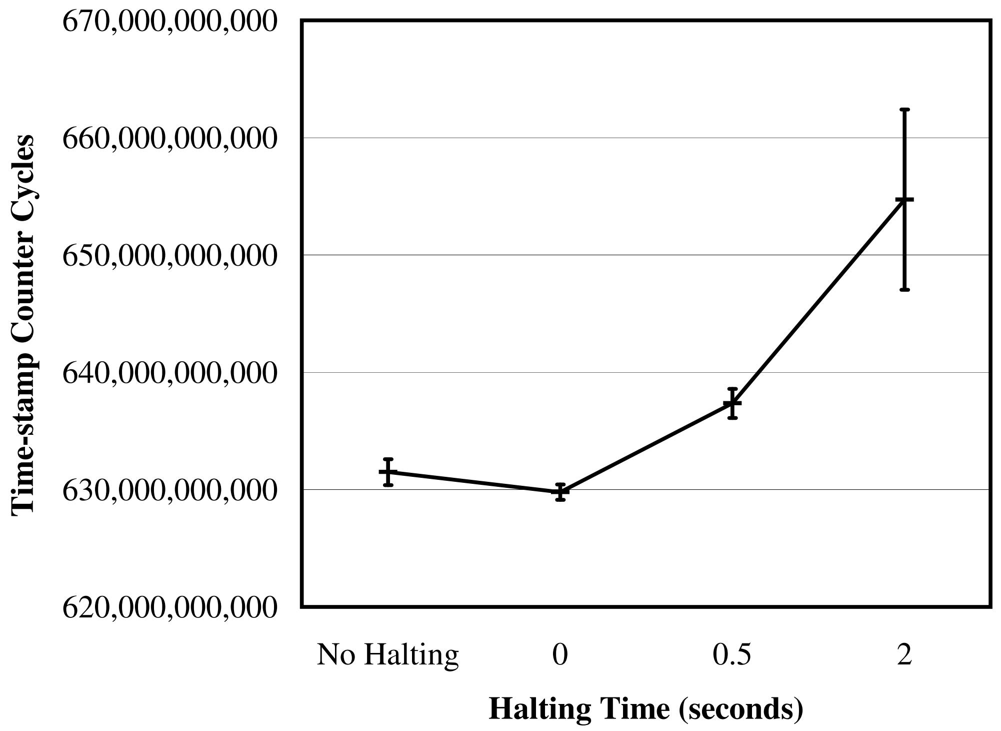

Figure 3.2: A statistically significant increasing trend. Notice that the axes do not cross at the origin; the scale has been chosen to allow the reader to see the shape of the trend, not the magnitude. The total variation here is less than 5%.

Increasing

Figure 3.2 shows the confidence intervals for the time-stamp counter running the eon benchmark with the 1/8 second halting rate. It is clear that there is a statistically significant difference between not halting the machine and halting it for long periods of time. The value of the time-stamp counter increases as the halting period increases, and the size of the confidence interval grows.

Steps

Figures 3.3, 3.4, and 3.5 are the confidence intervals running the parser benchmark with a halting rate of 1/8 of a second for ITLB misses, ITLB miss rate, and non-bogus instructions retired, respectively. It is clear in Figures 3.3 and 3.4 that there

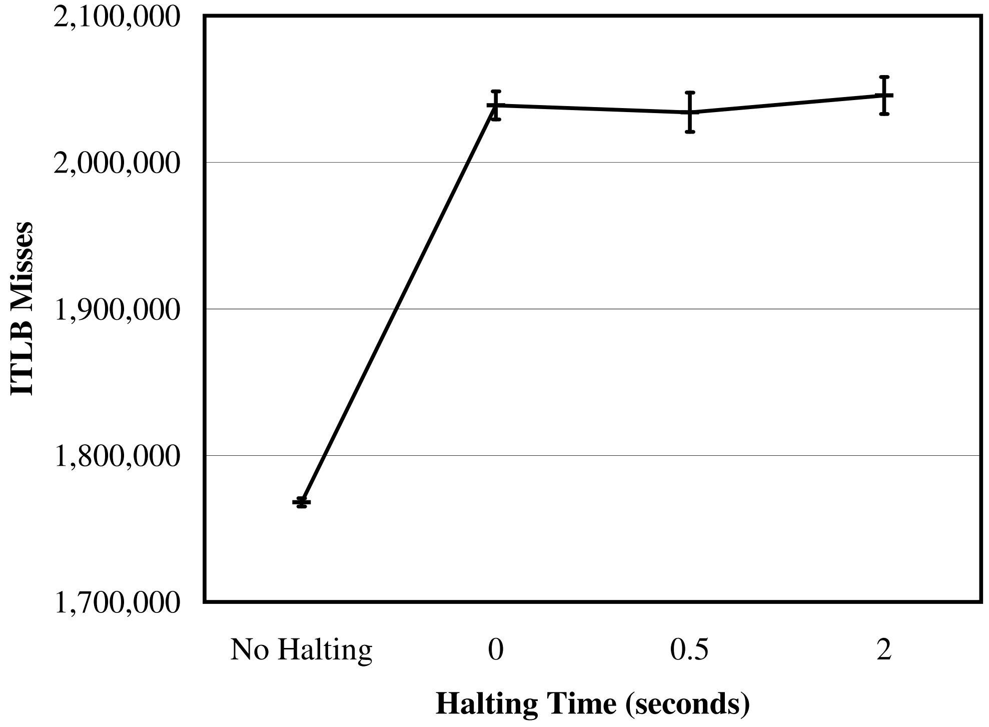

Figure 3.3: A step trend. This trend is one of the most common trends during the experiments. Notice the axes do not cross at zero, in order to present the shape clearly to the reader. See Figure 3.4 for an idea of the magnitude of the change.

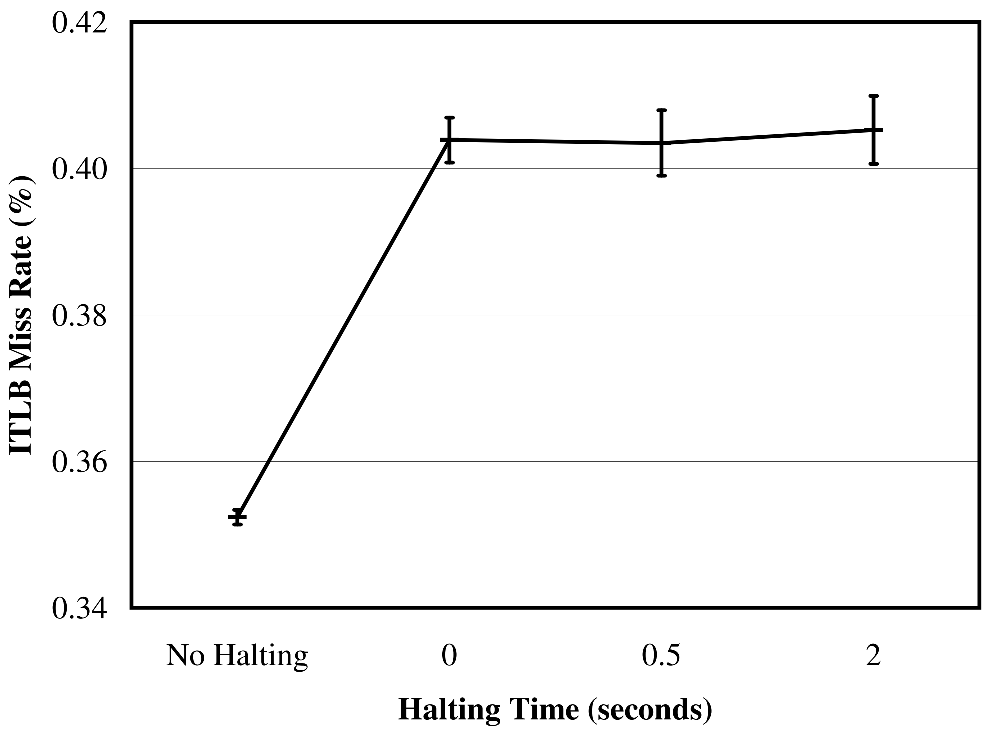

Figure 3.4: The same step trend as shown in Figure 3.3. This figure was included to help the reader see the impact on miss rate of the increase in misses.

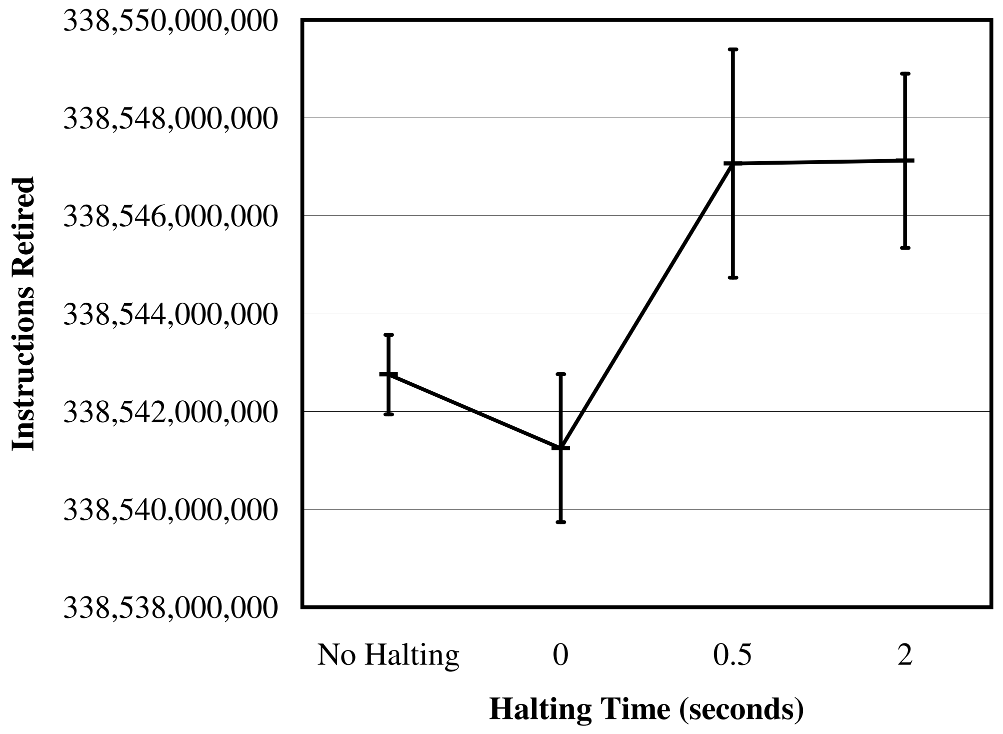

Figure 3.5: This is a second kind of step trend that occurs frequently in the data. There is no statistical difference in the number of instructions executed in this experiment when running without halting and halting for very short lengths of time, but halting the machine for any length of time increases the number of instructions executed. Notice that the total variation is less than 0.01%.

is a difference between halting and not halting the processor, but not between any two halting times. Figure 3.4 was included to give an idea of the impact of the extra ITLB misses. Figure 3.5 shows a significant difference between halting the processor for one half second or longer and not halting the processor or halting it for a very short period of time.

Statistically Unaffected Counters

Figure 3.1 presents the confidence intervals for memory stalls caused by a full reorder buffer running the parser benchmark with the same halting rate. Note that the mean of each confidence interval is contained by the confidence interval of the unhalted runs. Therefore there is not a statistically significant difference in the means.

3.2 Effects of Increasing Halting Times

We discuss the effects of increasing halting times starting with which counters were unaffected, moving to counters that were affected by halting, but not affected by increasing the halting time, and then counters that were impacted by increasing halting times.

Unaffected Counters

The three counters that were never significantly affected by increasing halting times were split loads, memory stalls caused by a full store buffer, and the cycle counter we used to sample the benchmark. The cycle counter based on global power events being unaffected means that there was not a statistically significant amount of extra processing time needed to complete the benchmarks as we increased the halting time. There was not a statistically significant difference in split loads or memory stalls caused by halting times at the granularity of halting for this experiment.

Counters Showing Steps

Counters that show a step trend as in Figures 3.3 and 3.5 are not affected by increasing the halting time, for the range we tested; however, we include them to differentiate from those where there was no significant difference. We list them in order of how many benchmarks showed significant steps. The following counters are listed from most to least affected: ITLB misses, loads retired, trace cache build mode, conditional branches, non-bogus instructions retired, all instructions retired, data TLB misses, 64K aliases, first level cache misses, and split stores. The amount each of these counters is affected varies from benchmark to benchmark, which we attribute to differences in the utilization of each resource.

The most consistently affected counter was ITLB misses. Figure 3.3 is a good example of this. The difference between the mean of the unhalted and halted values varies from benchmark to benchmark, but the step is consistent. This is due to the extra code involved in our interrupt service routine (ISR), which displaces ITLB entries that were used by the benchmark. We attribute the variability in the amount the counter is affected to its usage of ITLB entries. A related counter, ITLB reads and writes, was also affected in the same way.

Counters With a Trend

Two counters were affected in 16 out of 30 of the benchmark runs: the time-stamp counter and trace cache delivery mode counter, a statistically related counter that counts the number of cycles when the trace cache is in delivery mode. For the benchmarks where there is a marked increase in the number of cycles reported by these counters, there are no similar trends in other counters. Our first hypothesis was that the increased time had to do with second-level cache misses, which we have no way of counting. We therefore ran our misses benchmark stopping it for the same lengths of time as we had stopped the SPEC benchmarks. There was no significant trend in the value of the time-stamp counter. We also tried this with another microbenchmark, which spun in a tight loop similar to our interrupt timing loop. Again we found no significant trend.

It is interesting to note that there was no significant increase in the number of instructions executed or times halted. Since we interrupt the processor based on the number of cycles that the processor is not halted, there is an increase in the amount of time that the processor is in our ISR. In the bulk of our ISR interrupts are disabled, and the processor spins in a tight loop. One possible explanation is that our loop is the victim of thermal throttling, meaning that the processor was dissipating too much power and selectively disabled some resources. This would explain why our loop took longer to execute, but only with some benchmarks.

Halting Time Summary

The effects of increasing the halting time are very small in general. In fact, it only appears to significantly affect the total number of cycles that the benchmark is in the ISR, which is of no importance to trace collection. The counters that exhibited a step will be explored further in the next section on the effects of halting frequencies.

3.3 Effects of Increasing Halting Frequencies

For comparing varying halting frequencies, we again tallied the number of times that the counters were significantly affected. In this case there were no unaffected counters, meaning that each counter was affected for at least one of the benchmarks. We start by explaining the three types of trends we see, then list the counters that were affected in each way.

Shape of the Trends

There were three types of trends that we see with the affected counters: two which change linearly after a threshold and the same step shape that we saw with the halting times experiment. We will present the first two and refer the reader to Figure 3.3 and the previous discussion for the third, because it only occurred in two cases. Figures 3.6 and 3.7 show the same linear trend plotted in two different ways. Figure 3.8 shows an example of the second type of linear trend.

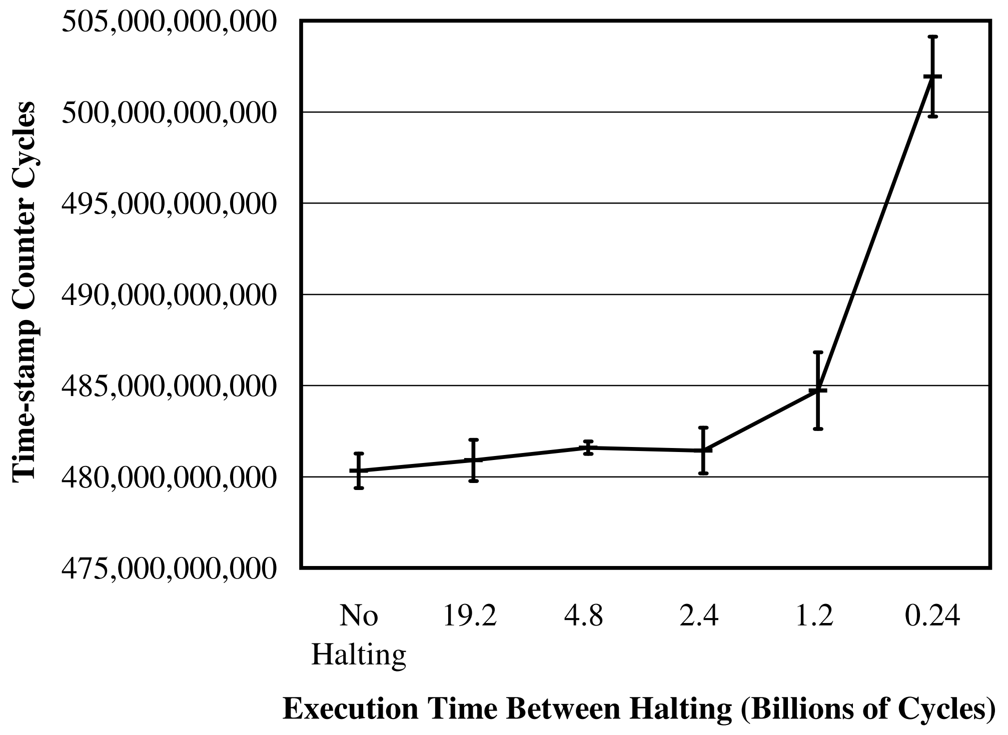

Figure 3.6: A linear trend plotted on a category axis. Figure 3.7 shows the same data in a scatter plot to make the linear trend more obvious. This was one of the more common trends for this experiment. Notice that the numbers on the X axis refer to the execution time between halting, and that the total variation here is about 5%.

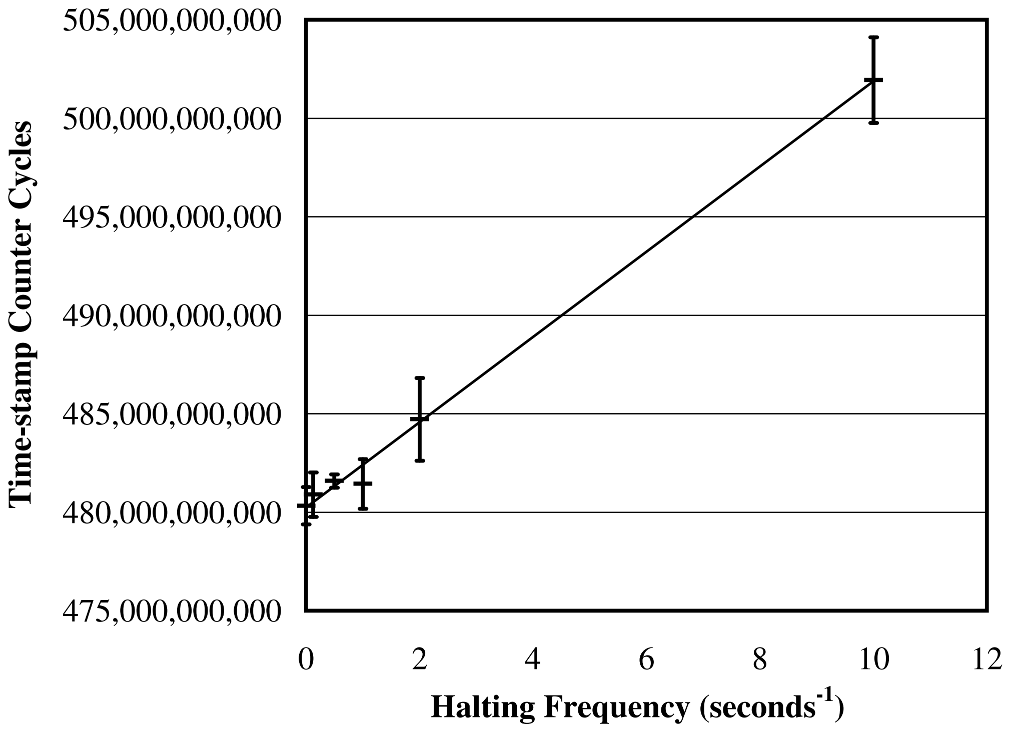

Figure 3.7: A Scatter plot of the data in Figure 3.6. This figure was included to make the linear relationship more easily visible. The line is the least squared error fit to the means.

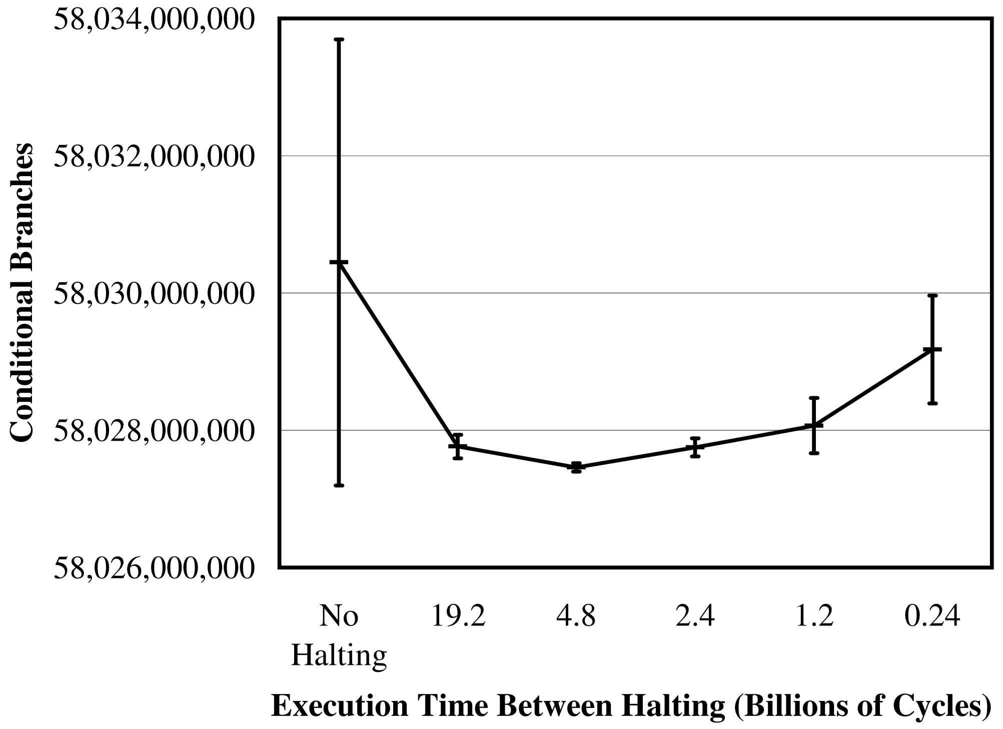

Figure 3.8: A linear trend with a large unhalted confidence interval. This graph appears much like Figure 3.6, but the large confidence interval of the unhalted run contains the means of the other runs. This means that there is no significant trend with the unhalted run included, but excluding the unhalted run gives a linear trend. This was a commonly observed pattern.

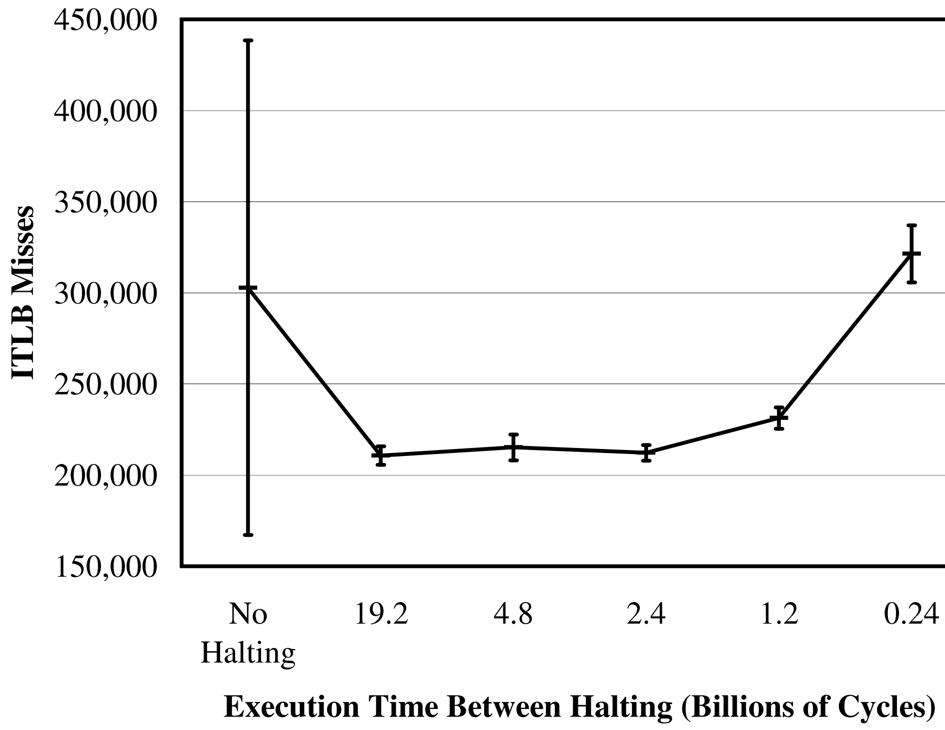

Figure 3.9: Another linear trend with a large unhalted confidence interval. This figure was included to illustrate the difference in the magnitudes that were observed in the trends. Compare the percentage variation of this figure to those of Figures 3.8 and 3.10.

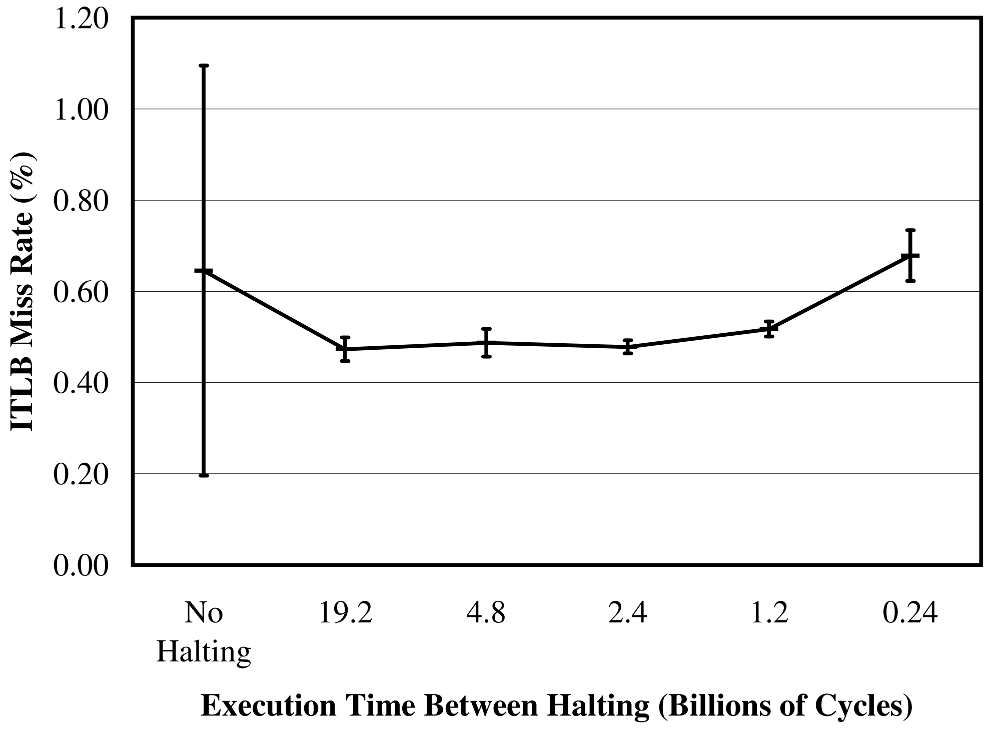

Figure 3.10: The ITLB Miss Rate graph corresponding to Figure 3.9. This figure was included to help the reader understand the magnitude of the difference.

Linear Trends

Figures 3.6 and 3.7 are two views of a good example of the most common trend that we see with affected counters. They show the confidence intervals around the mean for the time-stamp counter running the vortex benchmark, as the halting frequency increases. The numbers on the X axis represent the number of billions of cycles between halting. Note that there is no significant difference between the normal run and the halted runs until some point on the X axis, in this case when the processor is being halted every 1.2 billion cycles, or roughly one half second. After that point the difference increases approximately linearly with the number of times that the benchmark is halted. The point at which there is a significant difference varies from benchmark to benchmark, and from counter to counter, but most counters are not significantly affected until the frequency increases past halting once a second, or every 2.4 billion cycles.

Figure 3.8 shows an example of the next type of trend. This graph shows the confidence intervals around the mean for the values of the conditional branch counter running the gzip benchmark, as the halting frequency increases. It still shows the same linear trend as the number of halting times increases, but the confidence interval of the normal run is wider. This occurred for four of the counters when running gzip and two counters when running mcf; both of the benchmarks exhibited this behavior with conditional branches and non-bogus instructions retired.

It is interesting to note the effects of halting frequency on these counters in terms of percentage of the normal value. Figures 3.8 and 3.9 illustrate this. They are both from the same runs of the benchmark gzip, for conditional branches and ITLB misses, respectively. The difference between halting the processor every 1.2 billion cycles to every .24 billion cycles is over 40 percent of the mean value in the case of ITLB misses, but only $7 \cdot 10^{-11}$ percent for conditional branches. This was true for both types of linear trends.

Halting Frequency Summary

The most common trend for the counters is one that is linear with the number of times the benchmark is halted, i.e. inversely proportional to the halting frequency. We expect that with a high enough halting frequency this trend would not continue, because the benchmark would make so little progress in between interrupts. We believe that one tenth of a second is as fast as a reasonable tracing system would halt the processor; for tracing, the effect will be linear throughout the region of interest.

3.4 Results Summary

In our experiments, we have explored the effects of halting duration and halting frequency for all of the integer SPEC CPU2000 benchmarks, and three of the floating point benchmarks. The range of halting times and frequencies that we tested were chosen to represent current and near future tracing apparati.

The time the processor was halted had no significant impact on the counters during the time that the benchmark was running. Based on this result we expect no significant improvement in the quality of traces collected with tracing setups that can empty their buffers faster, within reasonable limits. It is, however, beneficial for those that would like their traces in a timely manner to increase the speeds at which the buffers may be emptied, see equation 2.1. Given that the time the processor is halted has no significant impact on the quality of traces collected, as measured by the performance counters, it is important to focus on halting frequency to improve tracing systems.

Many of the performance counters were significantly affected by increasing the halting frequency. The magnitude of these statistically significant differences varies substantially among counters and among benchmarks. Larger buffers will give more accurate traces in general, but most of the counters are not statistically affected if the processor is halted up to once every second. This means that a buffer size of 16 million references will be sufficient in the case that the bus is 100% utilized, or that our current system is sufficient for practical utilization on current processors.

Chapter 4

Conclusions and Future Work

4.1 Conclusion

In this thesis we used the performance counters of the Pentium 4 processor to measure the effect of halting the processor with various frequencies and with several halting times. We showed that the effects of halting are workload dependent, and that in general they increase linearly with the number of times the benchmark is halted. We also found that the length of time for which the processor remains halted does not significantly affect the performance counts in which we are interested.

Based on these results, we recommend building systems for capturing hardware traces which have buffers large enough to allow the machine to run for at least a second between halting to minimize the perturbation in the traces.

4.2 Future Work

Future work might include exploring ways to minimize the effects of halting and other uses for this methodology. Ways to minimize halting effects include compiling the halting driver into the kernel so that it maps to a TLB entry already used by the OS and defining a hardware-based halting mechanism that has very low overhead (perhaps based on a bus stall). Another use for this technique might be testing the warm-up time for caches. During execution of the benchmarks, you could compare flushing the cache to not flushing it using the counter values for the next section of execution.

Appendix A

Example Configuration Files

This appendix is included to give the interested reader some specifics on how to configure Brink and Abyss with XML configuration files. It consists of a brief overview of the file format and some examples from the configurations used in this work.

A.1 Brink Configuration Files

Configuration files are XML files that are divided into three sections: the processor identification, the program section, and the experiment section. The processor specifies which processor the configuration file was written for, the program section specifies which programs should be run, and the experiment section defines the experiments to be run on each program.

A.2 The Program Section

The program section contains a list of tags that associate names with commands. In Figure A.1, a2_gzip is the name given to the run of gzip that is initiated with the command in double quotes. The names for the programs must be unique, and they are run in lexical order. An important thing to remember is that any output generated by the program must be redirected so that it does not interfere with the brink and abyss tools, which communicate through standard in and out.

```
<programs>
        <a1_gzip command="runspec -n 1 -a run -I -l gzip>spec.out"/>
        <a2_gzip command="runspec -n 1 -a run -I -l gzip>spec.out"/>
                ...
        <a5_twolf command="runspec -n 1 -a run -I -l twolf>spec.out"/>
</programs>
```

Figure A.1: This excerpt from the program section of spec1-3.txt illustrates how to specify which programs will be run with Brink and Abyss.

A.3 The Experiment Section

The experiment section contains a list of tags that associate events with an experiment name. For each event, there is a base counter and the bits which must be set. In Figure A.2, every_0.3B_2s is the name of the experiment where clk_cycles is the name of the event used for EBS to halt the processor. The names of experiments and events must also be unique, and they are run in lexical order.

```
<e0normal>
        <clk_cycles base="global_power_events">
                <set> <running/> </set>
        </clk_cycles>
        <ld_miss_1L_retired_tag base="ld_miss_1L_retired"/>
        <ld_retired_tag base="loads_retired"/>
        <inst_ret_nbog base="instr_retired">
                ...
</e0normal>
<every_0.3B_2s>
        <clk_cycles base="global_power_events">
                <set> <running/> </set>
            <ebs type="imprecise" interval="300000000" buffer="2000"
                        sample="N" max="500"/>
        </clk_cycles>
        ...
<every_0.3B_2s>
```

Figure A.2: This excerpt from the experiment section of spec1-3.txt shows the way that ebs directives are used to halt the processor in experiment every_0.3B_2s and not in experiment e0normal

LIST OF REFERENCES

[1] J. K. Flanagan, B. Nelson, J. Archibald, and K. Grimsrud, “BACH:BYU address collection hardware, the collection of complete traces,” in *Proceedings of the 6th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Edinburgh U.K., Sept. 1992, pp. 128–137.

[2] K. Grimsrud, J. Archibald, M. Ripley, J. K. Flanagan, and B. Nelson, “BACH: a hardware monitor for tracing microprocessor-based systems,” *Microprocessors and Microsystems*, vol. 17, no. 8, pp. 443–458, Oct. 1993.

[3] J. L. Hennessey and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed. Morgan Kaufman Publishers, 1995.

[4] SIA, “International technology roadmap for semiconductors 2002 update,” 2002.

[5] A. J. Smith, “Cache memories,” *ACM Computing Surveys*, vol. 14, no. 3, pp. 473–530, Sept. 1982.

[6] R. A. Uhling and T. N. Mudge, “Trace-driven memory simulation: A survey,” *ACM Computing Surveys*, vol. 29, no. 2, pp. 128–170, June 1997.

[7] D. Burger, T. M. Austin, and S. Bennett, “Evaluating future microprocessors: The simplescalar tool set,” University of Wisconsin-Madison, Madison, Wisconsin, Tech. Rep. CS-TR-1996-1308, 1996.

[8] A. Srivastava and A. Eustace, “Atom: A system for building customized program analysis tools,” Digital (Compaq) (Hewlett-Packard) Western Research Labs, Tech. Rep. WRL Research Report 94/2, Mar. 1994.

[9] J. K. Flanagan, “Brigham young university trace distribution website.” [Online]. Available: http://traces.byu.edu/

[10] “Dinero cache simulator: Code, documentation,” 1997. [Online]. Available: http://www.ece.cmu.edu/ece548/tools/dinero/src/

[11] E. E. Johnson and J. Ha, “PDATS: Lossless address trace compression for reducing file size and access time,” in *Proceedings of the 13th IEEE International Conference on Computers and Communications*, Mar. 1994, pp. 213–219.

[12] E. E. Johnson, “Pdats II: Improved compression of address traces,” in *Proceedings of the 1999 IEEE International Performance, Computing, and Communications Conference*, Feb. 1999.

[13] A. Nanda, K. K. Mak, K. Sugavanam, R. K. Sahoo, V. Soundararajan, and T. B. Smith, “MemorIES: A programmable, real-time hardware emulation tool for multiprocessor server design,” in *Proceedings of the 9th International Conferece on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge MA, Nov. 2000, pp. 37–48.

[14] N. Chalainanont, E. Nurvitadhi, R. Morrison, L. Su, K. Chow, S.-L. Lu, and K. Lai, “Real-time l3 cache simulations using the programmable hardware-assisted cache emulator (pha$e),” in *Proceedings of the 6th International Workshop on Workload Characterization*, Austin,TX, Oct. 2003, pp. 86–95.

[15] B. Sprunt, “The basics of performance-monitoring hardware,” *IEEE Micro*, vol. 22, no. 4, pp. 64–71, July 2002.

[16] ——, "Pentium 4 performance-monitoring features," *IEEE Micro*, vol. 22, no. 4, pp. 72–82, July 2002.

[17] "PAPI: Performance application programming interface," Innovative Computing Laboratory, University of Tennessee. [Online]. Available: http://icl.cs.utk.edu/papi/

[18] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, , and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?" in *Proceedings of the 16th ACM Symposium of Operating Systems Principles*, Saint Malo, France, Oct. 1997, pp. 1–14.

[19] "Intel VTune performance analyzers," Intel. [Online]. Available: http://www.intel.com/software/products/vtune/

[20] *Pentium Pro Family Developer's Manual Volume 3: Operating System Writer's Manual*, Santa Clara, California, 1996, no. order no. 242692. [Online]. Available: http://developer.intel.com/design/pro/manuals/

[21] *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, Santa Clara, California, 2002, no. order no. 245472. [Online]. Available: http://developer.intel.com/design/pentium4/manuals/

[22] I. Gomez, L. Pinuel, M. Prieto, and F. Tirado, "Analysis of simulation-adapted SPEC 2000 benchmarks," *Computer Architecture News*, vol. 30, no. 4, pp. 4–10, Sept. 2002.

[23] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* New York: John Wiley & Sons, Inc., 1991.

[24] B. Sprunt, “The Brink and Abyss tools.” [Online]. Available: http://www.eg.bucknell.edu/~bsprunt/emon/brink“protect“unhbox“voidb@x“kern.06em“vbox–“hrul