

Does Halting Make Hardware Trace Collection Inaccurate? A Study Using Pentium 4 Performance Counters and SPEC2000

Myles Watson and J. Kelly Flanagan
Computer Science Department, Brigham Young University
Provo, Utah 84602
Email: myles,kelly@cs.byu.edu

Abstract—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.

I. INTRODUCTION

In this section we introduce the problem to be solved by stating that memory hierarchies help processors to increase performance even though main memory performance is increasing at a much slower pace. We then introduce 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 hardware performance counters.

A. Memory Hierarchies

The widening gap between CPU and main memory speeds makes memory subsystem design critical. There are many parameters to a memory hierarchy design, including 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. Choosing optimal values for these parameters is important since most of them contribute substantially to the cost of the hierarchy.

Researchers employ several methods to study the trade-offs among cache parameters, price, and performance. 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 exercise the design.

B. 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 [3]. Traces can be collected using functional simulation of the architecture [4], instrumenting the source or executable code of a benchmark [5], or connecting a hardware probe to the processor pins. The traces can then be stored in a raw format like BYU Address Trace Format [6] and Dinero [7], in a compressed format like PDATS [8], [9], or fed directly to a simulator without intermediate storage [4], [10], [11].

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 the implementations of tracing systems.

C. Trace Collection

In this work we are more concerned with trace collection than trace storage or trace processing. There are several methods of trace collection, including 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.

A hardware probe is designed to connect electrically to the processor pins and record the memory transactions. It allows the collection of address traces which include operating system and user references [1]. 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 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.

D. BACH: BYU Address Collection Hardware

BACH collects address traces using a hardware probe [1]. The current implementation of BACH uses Tektronix micro-

processor probes and logic analyzers, which 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. 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 [6].

E. Possible Sources of Error

BACH introduces some perturbation in traces that it collects. Three sources of error are bus activity due to the tracing device driver, the reordering of interrupt handlers, and displaced workload state. The extra bus activity can be removed with careful post-processing, but there is no way to correct for interrupt handler reordering or the changes in machine state, e.g. cache lines, which are used by the workload and displaced by the driver.

F. Outline

We submit that halting the processor introduces a negligible amount of error in trace-driven simulation with a sufficiently large tracing buffer. In order to make that case, we present experiments and their results which measure the differences in execution caused by halting the processor with varying frequencies and for various lengths of time. Section II describes the experiments we designed and details hardware performance counters. Section III presents our results, and Section IV gives our conclusions.

II. EXPERIMENTS

This section describes our experiments, hardware performance counters, and our experimental methods. 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, we fix the method of halting, and vary the time that 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.

1) *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 with replies there are at least six bus cycles per request, we have a maximum of 16 million requests per second. Our logic analyzer can buffer 8 million requests, and thus a half a second of requests at the maximum rate. We thus chose to halt the processor every 1.2 billion cycles, or approximately half a 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 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 I.

$$tx/f + x \tag{1}$$

| 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 I

EXPERIMENTS FOR DETERMINING THE EFFECT OF HALTING FOR DIFFERENT LENGTHS OF TIME. WE CONSIDER FEWER HALTING TIMES WHEN HALTING EVERY 1/8 SECOND TO REDUCE RUN TIMES.

2) *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 which 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 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 II 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 II

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.

A. Hardware Performance Counters

Modern microprocessors are complex. Their decoupled out-of-order and speculative execution engines make it difficult to predict how well specific code will run [12]. 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 which allow the programmer to access performance counts [13]–[16].

B. Performance Monitoring in the Pentium 4 Processor

The Intel Pentium 4 processor has more counters available for simultaneous event counting than its predecessors [17], [18]. 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.

Event based sampling (EBS) is another profiling method which 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 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.

We use EBS to interrupt the processor based on clock cycles. Our interrupt handler spins in a tight loop to simulate the time needed to empty the tracing buffer. 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.

C. Counter Selection

We started with the counters used by Gomez et al. [19]. In their work they use the SPEC CPU2000 benchmarks to compare benchmark reduction methods. 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, which is the preferred method for counting active clock cycles. We use this count with EBS to halt the processor based on benchmark execution time.

```
for (y=0;y<10000000;y++)
{
    for (a=0;a<MISS_TIMES;a++)
        x += buffer[a*0x2000]; //Miss in the Cache
    a--;
    for (z=0;z<64-MISS_TIMES;z++)
        x += buffer[a*0x2000]; //Hit in the Cache
}
```

Fig. 1. This is an excerpt from SimpleMisses.c, a small program which behaves differently in the data cache depending on the value of MISS_TIMES.

1) *Misses Benchmark*: In order to choose the remaining counters, and familiarize ourselves with the tools, we wrote some simple programs. These programs consisted of simple loops which generated an event to be counted. One of these programs was written to miss the caches different amounts of times. It consists of nested for loops which march through memory accessing locations which map to the same line, 32k apart. Figure 1 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 the two inner loops execute a total of 64 times.

2) *Cache Misses and Bus Transactions*: Figure 2 is a scatter plot with the value of 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, as expected. We used the same program to test L2 miss counts, but the counts were not correlated with MISS_TIMES. There are two different ways which the Intel manual [18] 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, which we wanted to avoid.

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, including those generated by prefetches. 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.

3) *64k Aliasing*: When trying to find L2 correlations, we noticed that elapsed time didn't 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 which depended on that value must be canceled and restarted. We decided to include it in the counters which we use.

4) *Allocate Remaining Available Counters*: From this point, we allocated the remaining counters. We added split loads, split stores, memory stalls due to the reorder buffer being full, and trace cache mode counters. There are not enough documented events to use all the counters in one of the groups, so we end up using 16 performance counters and

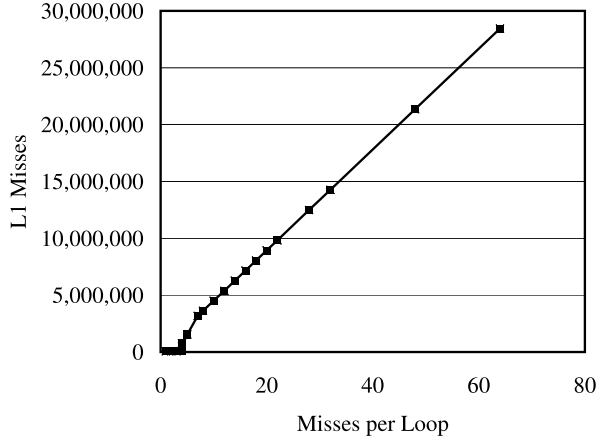


Fig. 2. First level data cache misses from SimpleMisses.c for several values of MISS_TIMES. Notice that when the associativity of the first level cache is exceeded, the number of misses increases linearly with the value of MISS_TIMES

the time-stamp counter, see Table III.

| 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 III

THESE ARE THE FINAL COUNTERS WHICH WERE SELECTED TO MEASURE THE PERTURBATION OF HALTING. THEY WERE SELECTED BECAUSE THEY MEASURE IMPORTANT PERFORMANCE EVENTS, AND/OR COULD BE COLLECTED SIMULTANEOUSLY.

D. Statistical Methods

We use statistical methods from the performance analysis book by Raj Jain [20]. In particular we make use of confidence intervals around the mean and an equation to determine necessary sample sizes.

1) *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. For example, 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.

Confidence intervals are computed using the sample mean, standard deviation, number of samples, and the t distribution for a given confidence level and number of samples. Equation 2 gives the formula for confidence intervals. We decided to use 95% confidence intervals before running the experiments.

$$(\bar{x} - sz_{1-\alpha/2}/\sqrt{n}, \bar{x} + sz_{1-\alpha/2}/\sqrt{n}) \quad (2)$$

2) *Determining Sample Size:* Formula 3 yields the number of samples necessary to find 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, \bar{x} is the sample mean, and r is the percent accuracy required. We decided to keep the variability of the counter values for unperturbed runs below 5%. We found that we needed at most 3 runs to do this for the integer SPEC CPU2000 benchmarks, and decided to use 5 in case the variability increased when the system was halted.

$$n = (100zs/r\bar{x}) \quad (3)$$

E. Experimental Environment and Details

Our Experiments are run on a Dell Optiplex GX260 with a 2.4 GHz Pentium 4 processor and 512 MB of RAM. We 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.

1) *Software:* The Brink and Abyss tools [21] provide us with an interface to the Pentium 4 processor's performance counters. They consist of a Perl script (Brink), a C program which interfaces to the device driver (Abyss front end), and the device driver which accesses the counters (Abyss device driver). We modified them slightly to allow us to use EBS for halting.

2) *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. 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.

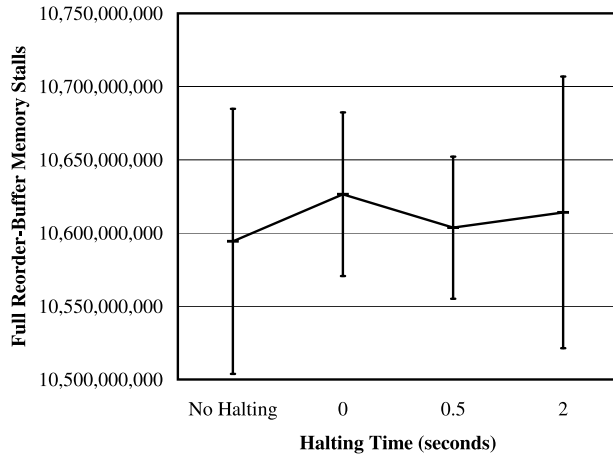


Fig. 3. No statistically significant trend. This figure contains no trend, because the confidence intervals contain each other's means. Therefore, even though there is variation, it does not indicate a pattern.

III. RESULTS

In this section we discuss our experimental results. We start by describing and presenting graphical examples of statistically significant trends. We then describe the trends which were encountered in our experiments. 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 section we urge the reader to pay attention to the scales of the graphs, as they were chosen to emphasize the trends, not the magnitude of the counts.

A. 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 which showed statistically significant patterns, and tabulated the results. Figure 3 is an example where there is no statistically significant effect of increasing the halting time, and is included for contrast with the figures 4 and 5, which show examples of statistically significant effects.

1) *Increasing*: Figure 4 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.

2) *Steps*: Figures 5 and 6 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 Figure 5 that there is a difference between halting and not halting the processor, but not between any two halting times. Figure 6 shows a significant difference between halting the processor

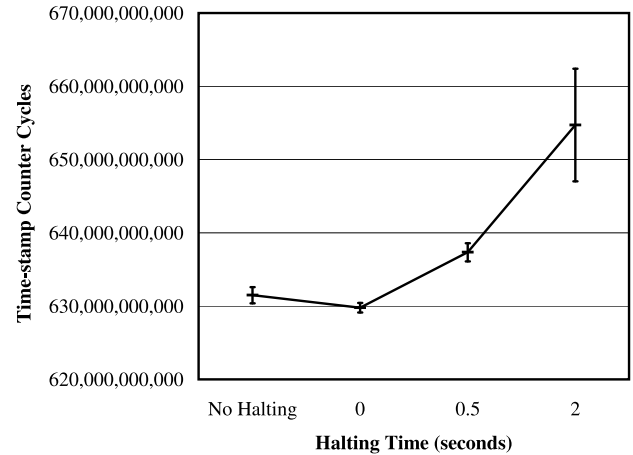


Fig. 4. A statistically significant increasing trend. Notice that the axes do not cross at the origin; the scale has been chosen to allow the reader to see the shape of the trend, not the magnitude. The total variation here is less than 5%.

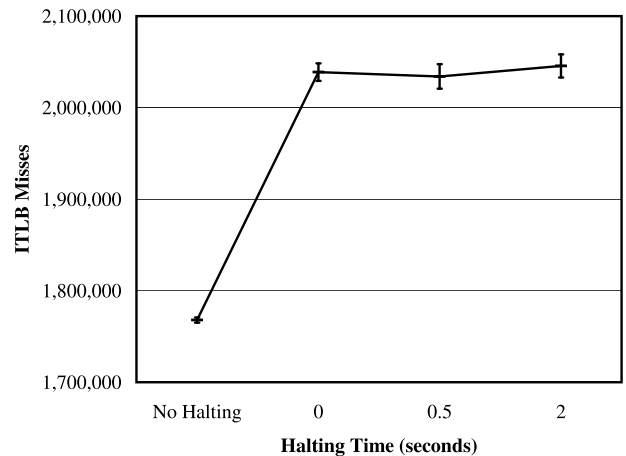


Fig. 5. A step trend. This trend is one of the most common trends during the experiments. Notice the axes do not cross at zero, in order to present the shape clearly to the reader. This variation in ITLB misses corresponded to 0.06% variation in ITLB miss rate.

for half a second or longer and not halting the processor or halting it for a very short period of time.

3) *Statistically Unaffected Counters*: Figure 3 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.

B. Effects of Increasing Halting Times

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

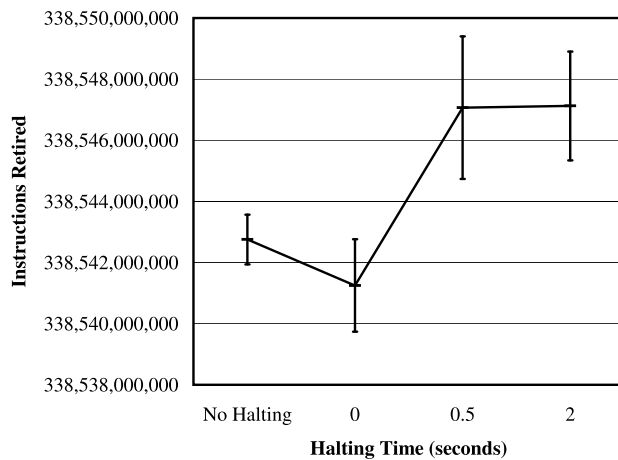


Fig. 6. This is a second kind of step trend which occurs frequently in the data. There is no statistical difference in the number of instructions executed in this experiment when running without halting and halting for very short lengths of time, but halting the machine for any length of time increases the number of instructions executed. Notice that the total variation is less than 0.01%.

1) *Unaffected Counters*: The three counters which 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.

2) *Counters Showing Steps*: Counters that show a step trend as in Figures 5 and 6 aren't 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 5 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 which 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.

3) *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 which 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.

4) *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 which exhibited a step were affected by halting, but not by increasing the halting times, and will be explored further in the next section.

C. 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 instead start by explaining the three types of trends we see, then list the counters that were affected in each way.

1) *Shape of the Trends*: There were three types of trends which 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 5 and the previous discussion for the third, because it only occurred in two cases. Figures 7 and 8 show the same linear trend plotted in two different ways. Figure 9 shows an example of the second type of linear trend.

2) *Linear Trends*: Figures 7 and 8 are two views of a good example of the most common trend which 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

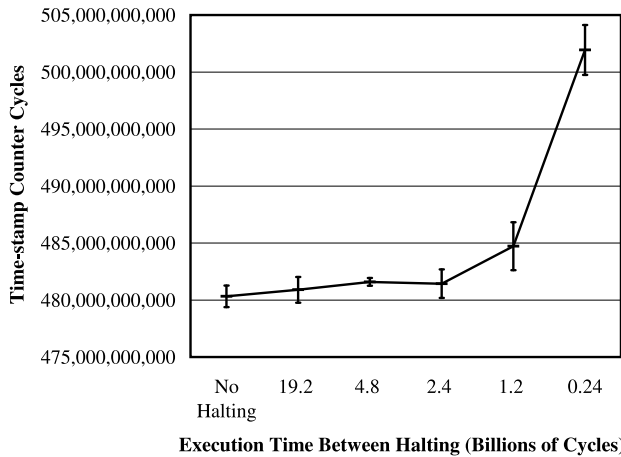


Fig. 7. A linear trend plotted on a category axis. Figure 8 shows the same data in a scatter plot to make the linear trend more obvious. This was one of the more common trends for this experiment. Notice that the numbers on the X axis refer to the execution time between halting, and that the total variation here is about 5%.

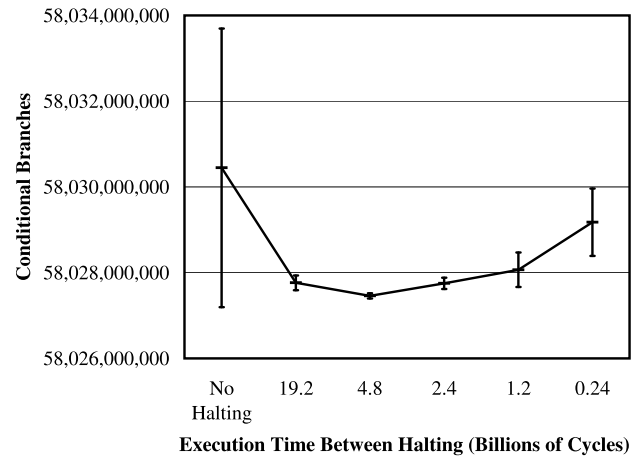


Fig. 9. A linear trend with a large unhalting confidence interval. This graph appears much like Figure 7, but the large confidence interval of the unhalting run contains the means of the other runs. This means that there is no significant trend with the unhalting run included, but excluding the unhalting run gives a linear trend. This was a commonly observed pattern.

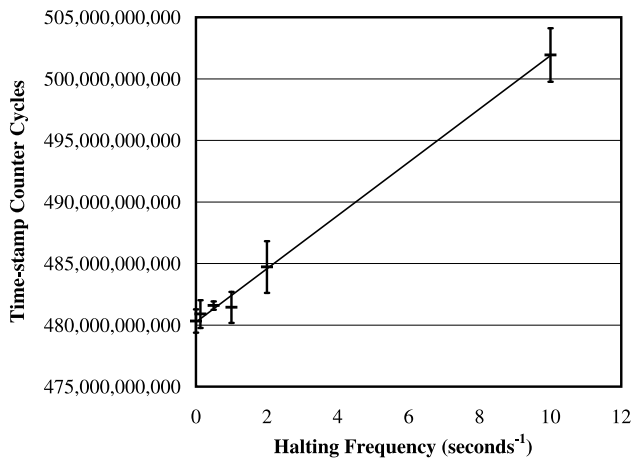


Fig. 8. A Scatter plot of the data in Figure 7. This figure was included to make the linear relationship more easily visible. The line is the least squared error fit to the means.

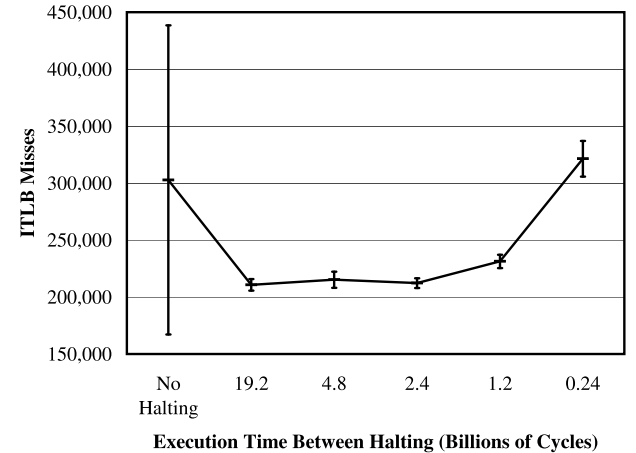


Fig. 10. Another linear trend with a large unhalting confidence interval. This figure was included to illustrate the difference in the magnitudes which were observed in the trends. Compare the percentage variation of this figure to that of Figure 9. The total variation of ITLB misses in this figure corresponds to a 1% variation in ITLB miss rate.

billion cycles, or roughly a 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 9 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 9 and 10 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.

3) *Halting Frequency Summary*: The most common way for the counters to be affected is linearly with the number of times the benchmark is halted, i.e. inversely proportionally

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.

D. 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 which we tested were chosen to represent current and near future tracing apparatus.

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 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 differences varies substantially among counters and among benchmarks. Larger buffers will give more accurate traces in general, but most of the counters aren't 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 the current implementation of BACH is sufficient for practical utilization on current processors.

IV. CONCLUSION

In this work 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 that the benchmark must be 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.

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] 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.

[4] 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.

[5] 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.

[6] J. K. Flanagan, "Brigham young university trace distribution website." [Online]. Available: <http://traces.byu.edu/>

[7] "Dinero cache simulator: Code, documentation," 1997. [Online]. Available: <http://www.ece.cmu.edu/ece548/tools/dinero/src/>

[8] 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.

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

[10] 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 Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, Cambridge MA, Nov. 2000, pp. 37–48.

[11] N. Chalanant, 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 (phaSe)," in *Proceedings of the 6th International Workshop on Workload Characterization*, Austin, TX, Oct. 2003, pp. 86–95.

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

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

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

[15] 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.

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

[17] *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/>

[18] *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/>

[19] 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.

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

[21] B. Sprunt, "The Brink and Abyss tools." [Online]. Available: <http://www.eg.bucknell.edu/~bsprunt/emon/brink/abyss/brink/abyss.shtm>