Understanding Measurement Perturbation in Trace-based Data

Todd Mytkowicz Amer Diwan
University of Colorado at Boulder
{Todd.Mytkowicz,diwan}@colorado.edu

Matthias Hauswirth
University of Lugano
Matthias.Hauswirth@unisi.ch

Peter F. Sweeney
IBM T.J. Watson Research Center
pfs@us.ibm.com

Abstract

Performance analysts commonly use trace-based data containing hardware and software metrics to understand performance. The trace data is generated by instrumenting the code to increment a counter when an event occurs and to collect hardware and software metrics in a trace. Unfortunately, the act of collecting a trace can perturb the behavior that the trace is trying to capture.

In this paper, we gain an understanding of perturbation due to measurement instrumentation of the system. We identify two mechanisms to quantify perturbation: inner and outer perturbation. Using inner perturbation, a performance analyst can determine when a run is perturbed by collecting too much information. Using outer perturbation, the performance analyst can determine if she can use the data from multiple runs as if the data were all from a single run.

Our evaluation of these mechanisms lead to two results. First, we are surprised to find that even with minimal instrumentation overhead, which increased instructions executed by less than 3%, high perturbation resulted, which prevented one from correctly reasoning about metrics within a trace or across traces. Second, the instrumentation of different software metrics interact in subtle, and not always obvious, ways making the impact of instrumentation on perturbation difficult, if not impossible, to predict.

Finally, we outline a methodology for collecting data while avoiding perturbation. When inner perturbation occurs, the performance analyst can spread out the data collection over multiple runs. When outer perturbation occurs, she can try different strategies for spreading out the data collection over multiple runs.

1. Introduction

In prior work [6, 5] we showed that to understand the performance of a modern system one needs to collect the traces of hundreds of software and hardware metrics. Hardware features, such as hardware performance monitors, and software tools, such as ATOM [14] and PIN [7], enable performance analysts to easily collect these metrics in traces. However, the performance analyst does not know if the data collection has perturbed the behavior that she was trying to understand. This is known as the observer effect. Most performance analysts assume that if the instrumentation does not change the execution time or instruction count by a large amount (say 5-10%) then the perturbation will be small. To our knowledge, no one has thoroughly explored the nature of perturbation and whether or not a small change in instructions or cycles actually guarantees that the perturbation is also small. This paper demonstrates an approach for measuring perturbation and also describes an approach for avoiding unacceptable perturbation.

Hardware support enables us to collect hardware metrics with minimal perturbation: counting the number of times a hardware metric occurs is done in hardware. However, collecting software metrics may significantly perturb the system. To observe the events counted by software metrics the system needs to be instrumented with additional instructions. This *instrumentation overhead* can affect almost all aspects of performance. For example, instrumented software may incur different misses in the caches and page tables, because the code is larger and additional loads and stores execute at runtime to increment counters. However, we can try to detect when there is significant perturbation and work around it. This paper presents such a methodology and also presents a preliminary evaluation of parts of the methodology.

In our methodology, a performance analyst decides on a set of metrics to collect. The selection of these metrics may be targeted (e.g., focus on the behavior of a particular aspect of the system) or may be general (e.g., the system behaves poorly and the performance analyst does not know why, so she wants to collects all the metrics that she can). In both cases, it is possible that the collection of the metrics will perturb system behavior; we present data that demonstrates that even low instrumentation overhead (in terms of percentage increase in the number of instructions) can dramatically perturb behavior.

To assist the performance analyst in collecting these metrics, our methodology informs the user when data collection has excessively perturbed a run. The performance analyst can start by collecting all metrics in a single run and use our method to determine if the perturbation is excessive. If so, she can partition the set of metrics into subsets and collect only one subset of metrics in each run. In prior work [5, 11], we have demonstrated an approach for aligning traces; we can use that alignment approach to enable the performance analyst to reason across the different subsets as if they had all been collected in the same run. Unfortunately, trace alignment does not always work; for this reason we also describe a method that detects when perturbation causes trace alignment to fail. In this case the performance analyst can try to collect different subsets and try again.

Our results indicate that low overhead (in instructions executed) does not necessarily translate to low perturbation. In some of our benchmarks, adding instrumentation amounting to fewer than 3% of total instructions dramatically perturbs the relationship between the metrics and thus affects a performance analyst's ability to effectively reason with the data.

The remainder of this paper is organized as follows. Section 2 introduces our methodology. Section 3 demonstrates that even the low overhead caused by instrumentation for software metrics can significantly perturb system behavior. Section 4 discusses what

NSF Next Generation Software Program Workshop '07, March 25-26, 2007, Long Beach, CA. 1-4244-0910-1/07/$20.00 2007 IEEE.

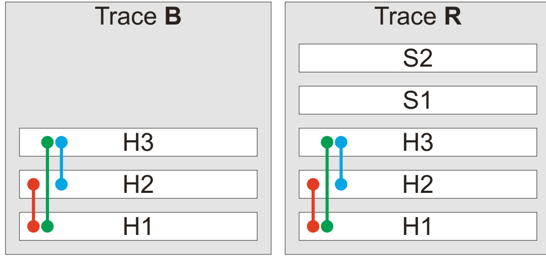

Figure 1. Inner perturbation

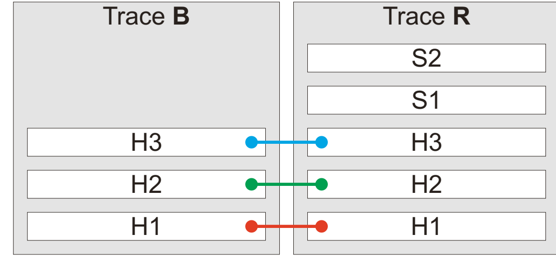

Figure 2. Outer perturbation

further work needs to be done to make our methodology a reality. Section 5 discusses related work and Section 6 concludes.

2. Methodology

We consider perturbation to occur when instrumentation changes relationships between metrics. For example, if before instrumentation L1 and L2 cache misses follow the same trends but after instrumentation they do not follow the same trends any more, we consider perturbation to have occurred. To capture the notion of "same trends" or "different trends" we use statistical correlation. Thus, in this paper, we quantify perturbation as the variation in correlation scores before and after instrumentation.

Our methodology provides two approaches to measure perturbation: *Inner perturbation* for deciding whether a given run has been perturbed by the metrics that it collected, and *outer perturbation* for deciding whether two runs are similar enough that we can reason across them.

2.1 Establishing a baseline

Before we can determine whether or not there is perturbation, we first need to establish a base line: what does an unperturbed run look like? Unfortunately, it is impossible to collect data without any perturbation at all. However, we expect that the perturbation with hardware performance monitors is minimal because the only cost is in taking the interrupt at each interrupt interval. Because the interrupt interval is quite large, tens to hundreds of milliseconds (Section 3), we use a run with only hardware performance monitors as the *baseline* trace.

2.2 Inner perturbation

To determine if instrumentation has excessively perturbed a run R, we compare that run to a *baseline* run, B, which does not capture software metrics. We require B and R collect the same hardware metrics. R generally collects additional software metrics. We compute the correlation score of each distinct pair of hardware metrics in B and do the same for R. We call these computed correlations "inner correlations" because the two metrics being correlated come from the same trace. We then compare the correlation scores in B to the corresponding scores in R (which gives us the inner *perturbations*). If the correlation scores are significantly different between B and R then we know that the software metrics have perturbed R. For example, let's suppose B contains the hardware metrics (H1, H2, H3) and R contains these metrics plus software metrics (S1, S2) (Figure 1). To see if collecting S1 and S2 perturbed R, we compute the inner correlation scores of the following metric pairs for B and also for R: (H1,H2), (H1,H3), (H2,H3). Then we compare the correlation of H1 with H2 in B to their correlation in R, the correlation of H1 and H3 in the B to their correlation in R and so on, to determine if collecting (S1, S2) has perturbed the run.

2.3 Outer perturbation

Outer perturbation enables us to determine if instrumentation has perturbed a run to the point that it can not be aligned properly with another run. Once again assume that we use a baseline trace B as above (though that is not strictly necessary). We align B to the trace of a run of interest R using an alignment technique. Then, for each hardware metric, we correlate the hardware metric in B to the same metric in the aligned R (Figure 2). We call these calculations *outer correlations* because they correlate the signal of a metric from one trace with the signal of the same or another metric from *a different* trace. If there is minimal perturbation and the alignment works correctly, then the correlation scores should all be 1.0 (i.e., perfect correlation). Any significant deviation from 1.0 (which we call outer perturbation) indicates a perturbation.

Outer perturbation assumes a trace alignment technique that aligns the trace records in the two traces. Trace alignment may align a trace record in one trace with multiple records in the other trace. In this paper, we use dynamic time warping [1, 5] to align traces; however, one can use any alignment technique.

3. Experiments

We now measure and analyze the effect of perturbation due to instrumentation on the relationship between metrics. We quantify the "relationship between two metrics" as the Spearman's correlation coefficient between the metrics.

3.1 Infrastructure

To conduct the experiments in this paper, we added an extension to CIL [12] which instruments C programs. Our extension inserts two kinds of instrumentation: for low-level events and for high-level events.

For low-level events we insert calls to PAPI [2], a portable interface for accessing hardware performance monitors, at the beginning of a program's execution to identify the set of hardware metrics to be collected. The cost of counting how many times a hardware metric occurs is zero because it is done in hardware.

For the high-level events, we add instrumentation to the C program. For each kind of instrumentation, we add a global variable to the C program and increment that variable every time that event occurs.

To collect hardware and software metric values at regular intervals, we use the `setitimer` interface provided by the Linux kernel. We register a signal handler that collects events over a millisecond interval. The signal handler tells PAPI to stop counting events, then fetches the values of the metrics, writes them to disk, resets their values, and resumes counting. Although collecting metric values introduces perturbation to a program's execution, the perturbation is minimal as long as the timer interval is sufficiently large. We picked our timer interval to be on the order of tens to hundreds of milliseconds (more on this below), and carefully designed the code

executed by the signal handler to do the least amount of work in collecting hardware and software events.

Given a configuration file that describes what hardware metrics to collect and exactly what software metrics to instrument and collect, our extension to CIL instruments C programs to generate traces that contain both high-level and low-level events (i.e., vertical traces [6]). We have added a flexible mechanisms for specifying software metrics; for example, users can indicate that every update of a particular variable or every call to a particular function is an event.

3.2 Benchmarks

Table 1 describes our benchmark programs. Three of the benchmarks are from the SPEC CPU2006 suite [15] and one is the cycle-accurate architectural simulator from the SimpleScalar toolkit [3].

For this paper we collected approximately 700 traces. Our subsequent analyses worked not just on individual traces but on pairs of traces. Because our analyses are quadratic in time and since we needed to run them many times, we limited our traces to approximately 3000 trace records (i.e., for each trace the signal handler executed about 3000 times). To accomplish this goal, we (i) picked timer intervals for each benchmark that yielded traces that were approximately 3000 records and (ii) picked inputs that were realistic but not so large that we would have to use an excessively large timer interval.

3.3 Significance of perturbation

We expect that the perturbation due to the tracing of hardware performance monitors is minimal, because the only overhead is in writing values to disk at the end of each timer interval. Because the timer interval is quite large, tens to hundreds of milliseconds (see the last column in Table 1), we use a run with only hardware performance monitors as the *baseline* trace for each benchmark.

Both of *inner* and *outer* correlation detect perturbation when there is a *significant* deviation between the correlation scores in the baseline trace and the trace instrumented with software performance monitors. The definition of *significant* depends on how much variation one normally gets between identical runs of the baseline trace: even if there is no perturbation, runs may not yield identical time series due to non-determinism introduced by operating system activity, etc. If the deviation between the correlation score of the baseline trace and the correlation score of the instrumented trace is smaller than the deviation observed due to non-determinism, then the deviation is not significant; if it is larger, then it is significant.

Figure 3 shows the inner correlation for two pairs of metrics of the sjeng benchmark; other benchmarks or metric pairs yield similar graphs so we omit them. The x-axis identifies a trace (we collected a total of 26 baseline traces, all of which measured only the hardware metrics). The y-axis gives the inner correlation score between a pair of metrics within a trace. It should be noted that we are using Spearman's correlation coefficient, which ranges from [-1,1]. To make the graphs more readable, we always take the absolute value of the correlation coefficient, and only plot the range [0.3,1], as correlation scores below 0.3 are not very interesting in our analysis.

We see that for both pairs of metrics, the curve is largely flat; in other words, while there is some inter-trace variation, the variation is small. The inter-trace variation identifies system perturbation that is due to non-determinism in the underlying operating system and hardware. We can use this result to help understand if deviations in correlation scores are due to instrumentation perturbation. If we observe a deviation in correlation score which is greater than the variations in Figure 3 then we can be quite confident that the deviation is due to perturbation.

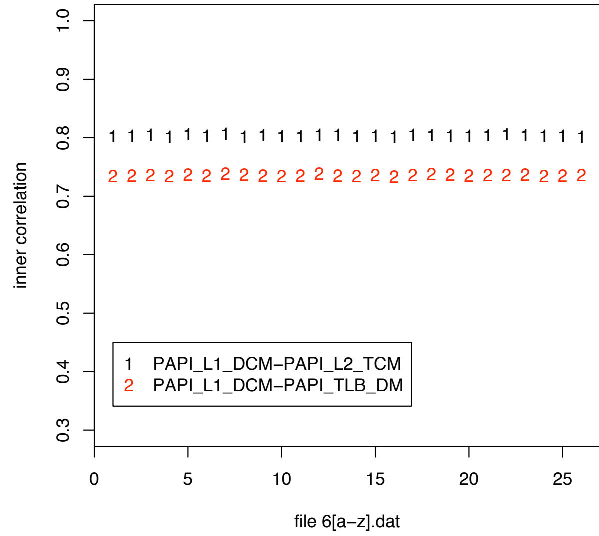

Figure 3. Inner correlation with zero instrumentation (baseline) in sjeng

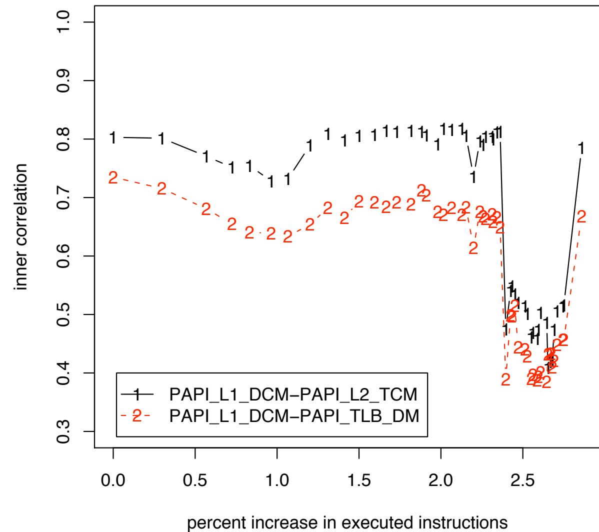

Figure 4. Inner correlation depending on instrumentation overhead in sjeng

3.4 Evaluation of inner perturbation

Figure 4 illustrates that instrumentation overhead can significantly impact the relationship between two metrics in the same trace file. In this figure, the x-axis is the percentage of instructions added by the instrumentation to collect software metrics. The y-axis is the inner correlation between two metrics. Each curve is the inner correlation between a pair of metrics; thus the shape of each curve

Benchmark	Description	Source	Input	Timer interval
bzip	Compression	SPEC CPU2006 [15]	input.source (ref input)	10ms
mcf	Vehicle scheduling	SPEC CPU2006 [15]	inp.in (ref input)	30ms
sim-outorder	Architectural simulator	Simplescalar [3]	gcc.ss -O integrate.i	100ms
sjeng	Chess	SPEC CPU2006 [15]	train.txt (train input)	200ms

Table 1. Benchmark programs

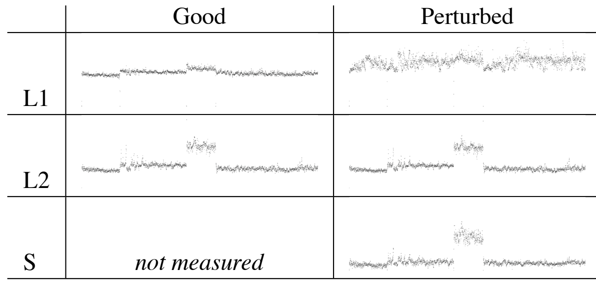

Figure 5. Perturbation due to instrumentation for one additional software metric in sjeng

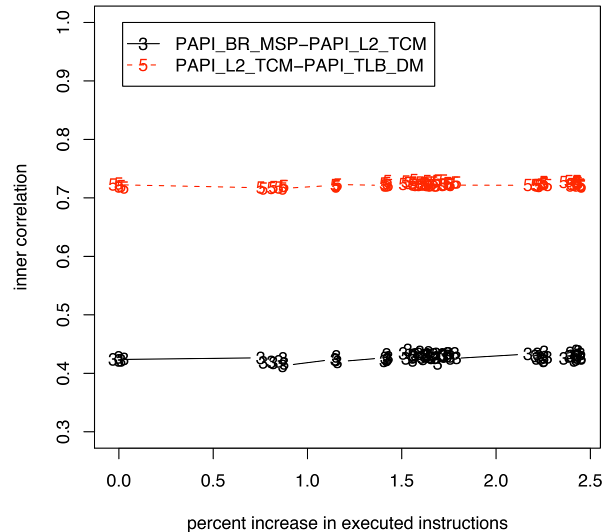

Figure 6. Inner correlation depending on instrumentation overhead in bzip

indicates how inner correlation changes as we add more and more instrumentation.

The leftmost point gives the correlation score for the baseline trace; i.e. the trace where we only collect hardware metrics. As a point of reference, note that this point corresponds to the first point in Figure 3. The next point in Figure 4 gives the correlation score when we collect one software metric, while the third point gives the correlation score when we collect two software metrics; and so on. For example, the trace file represented by the fifth point contains four software metrics: the three software metrics from the third point plus an additional software metric.

The figure illustrates, for the sjeng benchmark, that perturbation due to instrumentation significantly affects the relationship between metrics. The inner correlation for L1 data cache misses and L2 cache misses (`PAPI_L1_DCM-PAPI_L2_TCM`) starts at 0.8 and dramatically drops to less than 0.5 at around a 2.25% increase in instructions. The inner correlation of L1 data cache misses and TLB misses (`PAPI_L1_DCM-PAPI_TLB_DM`) has a similar, but slightly less dramatic change. Instrumentation overhead did not significantly perturb the hardware metric pairs that are omitted from the figure. Of the other benchmarks, sim-outorder exhibits similar behavior.

It is surprising that such a low instrumentation overhead (about 2.25% increase in instructions) causes such a dramatic perturbation. This goes against the common wisdom that if we perturb the instruction count or cycles by a small amount, the perturbation will also be small.

Figure 5 illustrates an explanation for the significant drop in Figure 4. It shows the signals of the two metrics involved in the `PAPI_L1_DCM-PAPI_L2_TCM` curve of Figure 4. Row *L1* shows `PAPI_L1_DCM` and row *L2* shows `PAPI_L2_TCM`. Column *Good* shows the signals for the last run before the drop in Figure 4, while column *Perturbed* shows the signals for the first run after the drop. The figure shows that the L1 signal changes drastically from *Good* to *Perturbed*. Row *S* shows the signal of the one additional software metric, `spm_func_search`, collected in the *Perturbed* run. For the *Good* run, the L1 and L2 signals are correlated. In particular, the bump in L2 in the middle of the signal is reflected in L1. For the *Perturbed* run, however, the L1 misses do not follow this bump as clearly. This phenomenon is reflected in the Spearman's correlation coefficient dropping to less than 0.5. The number of events counted by `spm_func_search` is particularly high during the bump. This is an indication that the additional instrumentation needed for collecting that metric causes the bump in the L1 signal to disappear.

Figure 6 illustrates that additional instrumentation overhead does not always lead to additional perturbation. The figure shows the inner correlation of two metric pairs in bzip. As the number of executed instructions increases by up to 2.5% (more than the increase needed to significantly perturb the inner correlation of sjeng), the inner correlation barely changes. The mcf benchmark is similarly unaffected by additional instrumentation.

3.5 Evaluation of outer perturbation

Figure 7 illustrates that instrumentation overhead can also impact the relationship between two metrics in different traces. The x-axis shows instrumentation overhead as the percentage increase in executed instructions, and the y-axis represents outer correlation. Each line is the outer correlation of a metric with itself, where one of the trace files is the baseline trace that only collects hardware metrics. As with Figure 4, each point represents a run that contains all software metrics from the point to its left plus one additional metric. Figure 7 gives the data for sim-outorder; sjeng and mcf exhibit similar trends.

The figure illustrates that the outer correlation for all pairs of hardware metrics are significantly affected by the increase in instrumentation overhead. The outer correlation of branch mispredictions with itself (`PAPI_BR_MSP`) is most affected, while the outer correlation of translation lookaside buffer (TLB) misses with itself

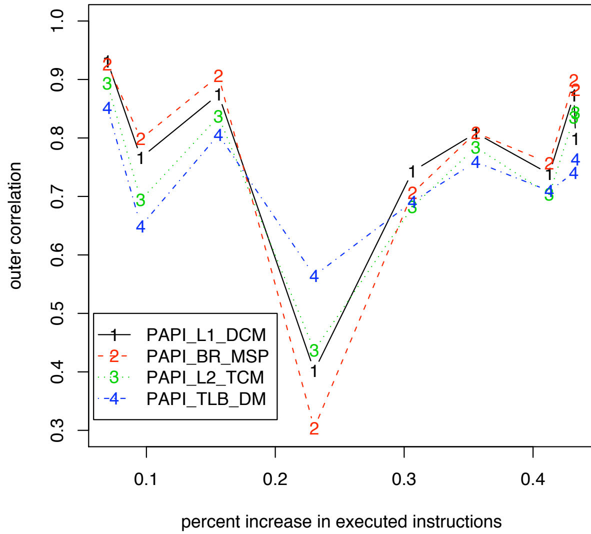

Figure 7. Outer correlation depending on instrumentation overhead in sim-outorder

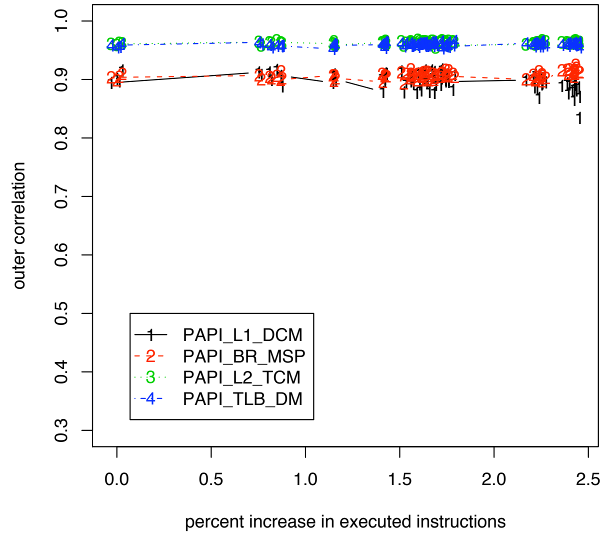

Figure 8. Outer correlation depending on instrumentation overhead in bzip

(PAPI_TLB_DM) is least affected. Branch mispredicts drops from a correlation of above 0.9 to 0.3 with only 0.2% increase in executed instructions. Less dramatically, TLB misses drop from a correlation of 0.85 to below 0.6. Interestingly, as we add software metrics after the drop, the outer correlation for the metrics recovers. This implies that the instrumentation of different software metrics interact in subtle, and not always obvious, ways.

This graph illustrates that low instrumentation overhead can radically change the relationship, from being highly correlated to being poorly correlated, between two metrics collected in different traces.

Figure 8 shows the outer correlation for a different benchmark, bzip. It illustrates that the outer correlation between the same hardware metric in two different traces does not always have to be impacted by the overhead of additional instrumentation. While the outer correlation in sim-outorder was significantly affected by an overhead of less than 1%, the outer correlation in bzip is largely unaffected even with an instrumentation overhead of 2.5%.

To summarize, our results suggest that perturbation does not depend on the number of instrumentation instructions that we execute. Thus, our technique for detecting perturbation is useful even if the instrumentation overhead is very small.

4. Discussion

The introduction described our methodology for collecting software metrics while keeping perturbation down. Section 2 described our approach for quantifying perturbation and Section 3 demonstrated our metrics with C programs. However, before our methodology is fully usable we have two open problems we need to address:

1. As we saw in Section 3 adding more instrumentation does not always increase perturbation; in some cases it may actually decrease perturbation. Thus, finding subsets of metrics to collect together without excessive perturbation is a non-trivial problem. The most general, but prohibitive, approach will try all possible partitionings into subsets. We will investigate greedy approaches to this problem.
2. Our mechanism for detecting perturbation assumes that if there is any significant perturbation it will perturb the relationship between two hardware metrics. This may not be the case in general: for example, a given perturbation may not affect relationship between hardware metrics but may affect the relationship between two software metrics or one software and one hardware metric. We do not know if this phenomenon happens in practice; we will investigate this further in the continuation of this work.

5. Related work

Prior work is rather limited in the field of trace perturbation, and can be broken into two distinct categories (i) perturbation *measurement* and (ii) perturbation *management*.

Indeed, our work falls into the category of perturbation measurement; we have provided an operational definition of perturbation, as we define a program to be perturbed if a set of hardware performance metrics *significantly* change, as a function of adding measurement instrumentation, and we have provided results that measure these fluctuations. Daigle et al[4] also provide an operational definition of perturbation when collecting program address references. Their approach looks at aggregate runtime slowdown as a function of the type of measurement instrumentation (*low* or *heavy*) as well as the overall increase in process memory. Using these metrics, the authors provided a means to describe when a program was dominated by measurement infrastructure. While useful for their performance analysis tasks, the metrics (runtime slowdown and memory increase) they used are far too coarse grained to accurately capture the true behavior of a program running on today's modern hardware.

Perturbation management is the process whereby the performance analyst realizes and accepts that any measurement infrastructure will necessarily perturb the program being measured. This approach builds models of perturbation incurred by an ob-

servation, and uses those models to factor out any measurement perturbation from the final trace presented to the performance analyst [9, 8, 10, 13]. This approach is limited by two main issues. First, the ability of one to recreate a *true* trace is only as good as the model one is able to build, and building an accurate model is incredibly hard. Indeed that is why [9] use a linear model of perturbation. Secondly, prior work only looked at overall program runtime as a means to measure perturbation. Our work here shows that even with a 2.5 % increase in instructions, underlying hardware metrics are severely perturbed, and thus these definitions of perturbation are too coarse grained.

6. Conclusions

It is impossible to measure a real computer system without perturbing its behavior. This is especcially the case when the measurement involves adding code to a system, which is the case when we want to collect software metrics.

As performance analysts, we realize and accept that we cannot eliminate perturbation, however, we can detect it. In this paper, we have identified two mechanisms to quantify perturbation: inner and outer perturbation. Using inner perturbation, a performance analyst can determine when a run has been perturbed by collecting too much information. Using outer perturbation, the performance analyst can determine if she can use the data from multiple runs as if the data was all from a single run.

Our evaluation of these mechanism lead to two results. First, we are surprised to find that even with minimal instrumentation overhead, which increased instructions executed by less than 3%, high perturbation resulted, which prevented one from correctly reasoning about metrics within a trace or across traces. Second, the instrumentation of different software metrics interact in subtle, and not always obvious, ways making the impact of instrumentation on perturbation difficult, if not impossible, to predict.

Finally, we outline a methodology based on these mechanisms to collect measurement data while avoiding perturbation.

7. Acknowledgements

This work is supported by NSF CSE-0509521, NSF Career CCR-0133457, DARPA contract NBCH30390004, and a gift from Intel. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

References

[1] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In *Working Notes of the Knowledge Discovery in Databases Workshop*, pages 359–370, July 1994.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, 2000.

[3] D. Burger and T. Austin. Simplescalar llc. http://simplescalar.com.

[4] R. Daigle, C. Xia, and J. Torrellas. Low perturbation address trace collection for operating system, 1996.

[5] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 281–296, New York, NY, USA, 2005. ACM Press.

[6] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: understanding the behavior of object-priented applications. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 251–269, New York, NY, USA, 2004. ACM Press.

[7] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM Press.

[8] Allen D. Malony. Event-based performance perturbation: a case study. In *PPOPP '91: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 201–212, New York, NY, USA, 1991. ACM Press.

[9] Allen D. Malony and Daniel A. Reed. Models for performance perturbation analysis. In *PADD '91: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging*, pages 15–25, New York, NY, USA, 1991. ACM Press.

[10] Allen D. Malony and Sameer S. Shende. *Overhead Compensation in Performance Profiling*, volume 3149/2004 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2004.

[11] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Aligning traces for performance evaluation. In *Next Generation Systems Workshop*, April 2006.

[12] George C. Necula, Scott McPeak, S. P. Rabul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of Conference on Compiler Construction*, pages 213–228, 2002.

[13] Sekhar R. Sarukkai and Allen D. Malony. Perturbation analysis of high level instrumentation for spmd programs. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 44–53, New York, NY, USA, 1993. ACM Press.

[14] Amitabh Srivastava and Alan Eustace. Atom: a system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, New York, NY, USA, 1994. ACM Press.

[15] Standard performance evaluation corporation. SPEC CPU2006 benchmarks. http://www.spec.org/cpu2006/.