We have it easy, but do we have it right?

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

We show two severe problems with the state-of-the-art in empirical computer system performance evaluation, observer effect and measurement context bias, and we outline the path toward a solution.

1. Introduction

To evaluate an innovation in computer systems, performance analysts measure execution time or other metrics using one or more standard workloads (e.g., the SPEC benchmarks). The performance analyst may carefully control the environment in which measurement takes place, or the *measurement context*, and repeat each measurement multiple times. Finally, the performance analyst may use statistical techniques to characterize the data. If the number of configurations of the system under study is large, the performance evaluation may take weeks or occasionally even months of *computer time* to collect. However, since we can often run different configurations and different workloads in parallel, performance analysts typically collect the data that they need for a paper in a few days of elapsed wall-clock time using using cheap commodity hardware.

In contrast the data collection process for natural and social sciences is much harder. Natural scientist often expend enormous effort and cost to design and conduct experiments that produce credible and repeatable data. A single experiment in many scientific domains can cost thousands of dollars and thus researchers expend significant effort to ensure that they actually get useful data from their experiments. Social scientists often expend months or more of human effort to collect a single data set. They make great effort to ensure that the data they collect is not tainted and use statistical techniques to identify significant trends in the raw data.

In other words, computer systems researchers have it easy. Our cost for collecting data is negligible compared to other scientists: we easily generate reams of data for *each* paper and it typically takes only a few days of wall-clock time. Unfortunately, while cheap, the data that we collect may or may not be actually useful. This paper shows how easy it is to produce poor (and thus misleading) data for computer research and suggests techniques for producing high-quality data. Unfortunately we find that there is no free lunch: to gather high-quality data, performance analysts need to work much harder than to collect poor-quality data.

What is "high quality" data? We define high-quality data as data that captures the true behavior of the system that we are evaluating; "poor quality" data is the dual of high-quality data. In this paper we explore two common sources of poor-quality data. First, we get poor-quality data if our data collection perturbs the behavior of the system that we are measuring; this is often known as the "observer effect". We show that even a seemingly insignificant measurement probe can dramatically alter system behavior; thus, perturbation is much more common than most performance analysts probably realize. Second, we get poor-quality data if we measure the system in a particular set of contexts and that set does not capture the range of reasonable contexts that a user of the system might encounter; this is known as "measurement context bias". We show that different contexts (e.g., settings of environment variables or compiler flags) favor different configurations of the system. Thus, the context can show a particular configuration in an optimistic or a pessimistic light; we cannot tell which one it will be in advance.

The remainder of the paper presents data on how easy it is to get poor-quality data and suggests ways of dealing with it.

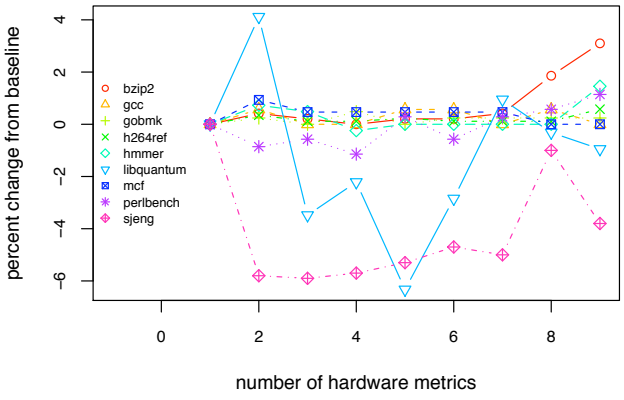

Figure 1. We see perturbation even if we collect only hardware metrics.

2. Poor-quality data due to the observer effect

Most approaches for measuring a system require us to insert probes into the system. Even if we collect data using hardware performance monitors (which the hardware automatically increments) we still need probes to set up counters and to read out the values of the hardware registers at selected points. We assume that if our probes are insignificant so are any perturbation effects that occur due to our observation. Section 2.1 shows that this assumption does not hold. Section 2.2 discusses how we can deal with this.

2.1 How large is the perturbation

Figure 1 presents the total cycle count when we collect only hardware metrics for SPECint 2006 benchmarks on a Pentium-4 workstation. We used PAPI [2] to collect data from the hardware monitors. We insert two set of calls to PAPI in our applications: one to

This work is suported by NSF CSE-SMA 0509521 and by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.
NSF Workshop at IPDPS, '08 April 13-14, 2008, Miami, CA.

set up the counting at the beginning of the program and one to read out the values of the counters at the end of the program. We need one call per hardware metric to set up the counters but only one call in total to read all the counters.

The leftmost point (x-axis label is 1) is the cycle count when we collect only one hardware metric, TOT_CYC, and is the baseline for subsequent points along the x-axis. Each subsequent point adds a new hardware metric. We confirmed each point in these graphs (and others in this paper) with three runs each since we wanted to make sure that our results were not an artifact of inter-run variation. We see that adding hardware metrics can significantly change the cycle count; our "benign" observation changes system behavior. In other experiments we established that with the exception of the *libquantum* benchmark, our programs exhibit minimal inter-run variation; thus the variation in Figure 1 is indeed due to measurements.

Figure 1 is surprising. How can such little observation change system behavior? The change in cycles does not come from the increments because the hardware increments the metrics without any cost. The culprit for the change in cycles must be the calls to set up the hardware monitors or the calls to read out the hardware monitors. It seems unlikely that a 6% change in the number of cycles can be directly due to a handful of calls in relatively long runs (our shortest run is over 13 seconds); in related work we have found that a readout call to PAPI is at most a few thousand instructions. Thus, we believe that the perturbation comes from indirect effects of the calls. Specifically, inserting these calls in the program affects code (and thus ultimately data) layout which can interfere with branch predictors, instruction caches, etc.

Our graph above shows that collecting only hardware counters can perturb data; **Figure 2** shows that the perturbation is even worse if we collect software counters; each counter measures some software event (e.g., number of insertions in the database) on a Pentium-4 workstation.

For each of the graphs, the x-axis gives the percent increase in instruction count from the baseline due to instrumentation needed to observe system behavior (baseline is the configuration with no instrumentation). Note that in the most extreme case we increase the instruction count by 2.0% (in mcf) and thus our instrumentation is not excessive: we would not expect instruction overhead to significantly perturb the system.

Each graph contains one curve for each frequently occurring hardware metric.[1] The y-axis gives the change in the total counts as a percentage of total counts with no instrumentation (i.e., the baseline). Note that the different benchmarks use different scales for the y-axis to best accommodate the total count data. We see that adding instrumentation for observation can dramatically perturb our runs.

We can draw two main conclusions from our results.

First, *perturbation due to observation is commonplace and dramatic*. With the exception of hmmer, the total counts of all benchmark changes dramatically as a result of adding instrumentation.

Second, *perturbation due to observation is unpredictable*. Specifically the amount of instrumentation has no bearing on the amount of perturbation. Even adding instrumentation to just read the values of hardware registers twice during an entire run can change total counts in either direction. This is totally counter intuitive. For example, we expected some kind of a monotonic relationship between amount of instrumentation and change in total counts; we see this is clearly not the case. Finally, we cannot predict perturbation in one metric using perturbation in another metric. We see many instances of this in our data. For example, in h264ref the change in total counts of TOT_CYC due to increment perturbation is relatively modest (8% between its lowest and highest point) but the change in RES_STL is dramatic (over 80%).

[1] By "frequently occurring" we mean metrics that occur at least once every 20 cycles in the original, uninstrumented program.

2.2 How to avoid the observer effect

We believe that there cannot be a general approach for avoiding the observer effect. Thus, performance analysts need to take measures, based on how they intend to use the data, to ensure that perturbation due to their observation does not mislead them. In this section, we describe the strategy we have used in our work; this is not an easy-to-use or a fully general strategy, but it works at least for our needs.

In our work on understanding the performance of Java systems, we need to collect traces of both software and hardware metrics. We reason over these metrics in an attempt to explain poor performance in some part of the program execution. If our measurements perturb the system, then we may arise at an incorrect conclusion about the cause of the poor performance.

To make sure that measurement perturbation is not leading us along the wrong path, we employ *validation* throughout our exploration. A validation step checks if our understanding of the system so far is correct by acting on the understanding and then confirming that the outcome is consistent with our understanding. In other words, we are proposing causality analysis which other scientific disciplines already contend with [10].

To see how this works, let's suppose we notice (by visualizing a trace) that at certain points in a program's execution there are a large number of cache misses. Analyzing our trace may lead us to believe that the representation of a particular data structure is resulting in a large working set size and thus cache misses. This belief may be correct or it may be an artifact of observation perturbation. To validate this belief, we change our data structure so that, at least for our test runs, it consumes less space (e.g., by using smaller arrays, because we know that our test runs can run with smaller arrays). We then regenerate the trace and confirm that the cache misses have changed accordingly. If they have changed accordingly then we have a strong reason to believe that our belief is correct; if they have not then we have a strong reason to believe that our belief is incorrect due to perturbation.

This approach does not avoid perturbation from observation, but tolerates it: if we reach the correct understanding about our system's performance then perturbation does not matter. Just because we have reached the correct understanding in this case does not mean that our measurements did not perturb our system; if we use the same data for some other task we may find the perturbation is large enough that it leads us to an incorrect conclusion. We have applied this methodology to all of our case studies [4].

3. Poor-quality data due to measurement context bias

In order to get reproducible and deterministic results, performance analysts typically control and fix the measurement context for a set of experiments. For example, performance analysts use the same version of the compiler, same compiler flags, same OS, same environment variables, lightly loaded machine, etc., to factor out the effect of the measurement context on their experiments. Underlying this methodology is the assumption that results they obtain in one measurement context carry over to other measurement contexts.

For example, let's suppose a researcher comes up with an innovation X to a system S. To demonstrate the benefit of X, the researcher can run both S and $S + X$ on standard workloads and compare the elapsed times for S and $S + X$ on each workload. If the elapsed times for $S+X$ are on average smaller than the elapsed times for S, then the researcher may conclude that X is a worthwhile innovation. However, this may or may not be the case. For

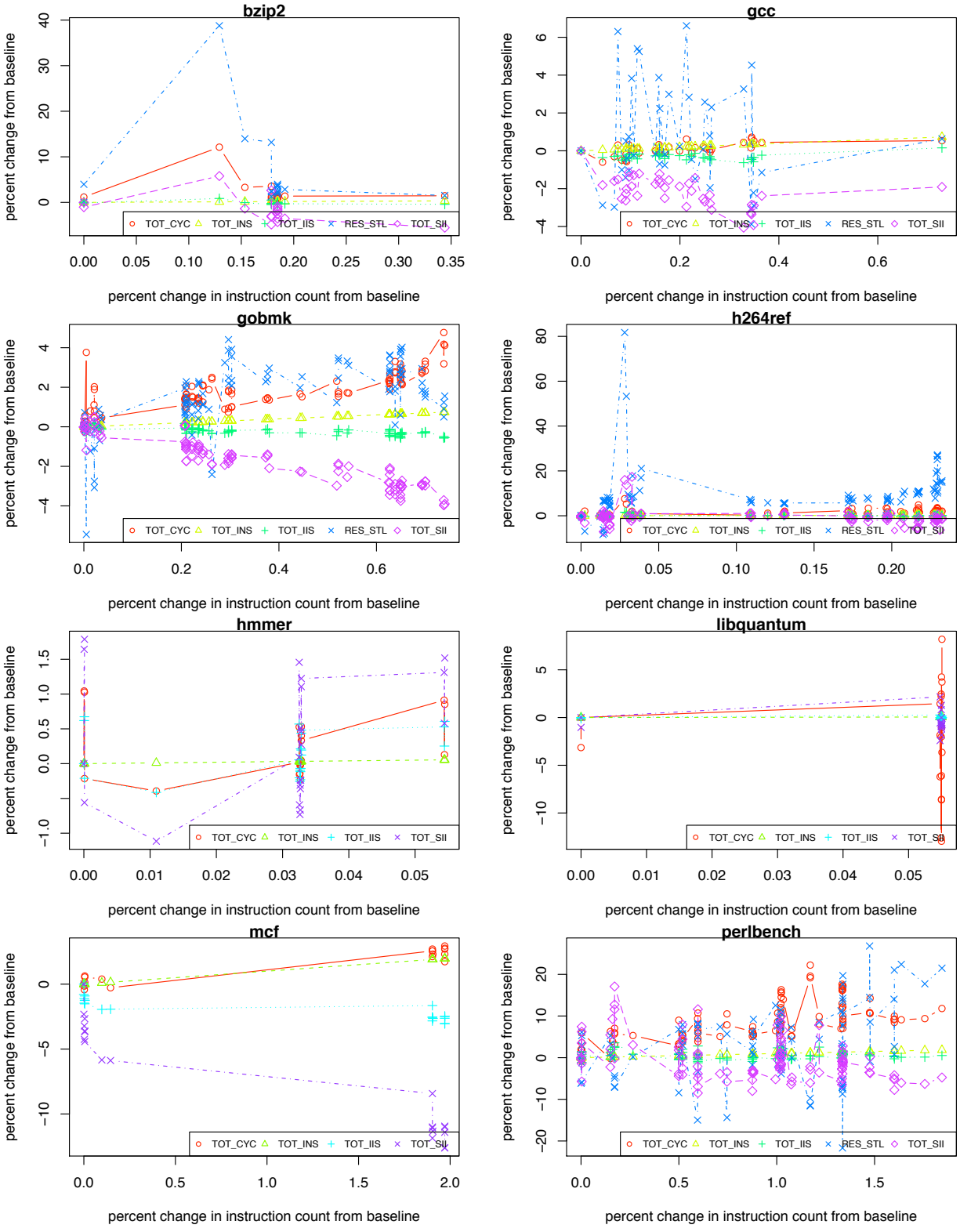

Figure 2. Increment perturbation and its effect on total counts and correlation

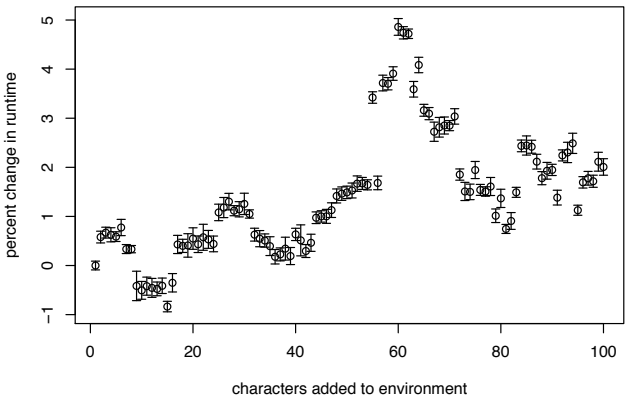

Figure 3. The mean and 95% confidence interval of the mean for the change in perlbench's run-time as an environment variable's value is changed on Pentium 4.

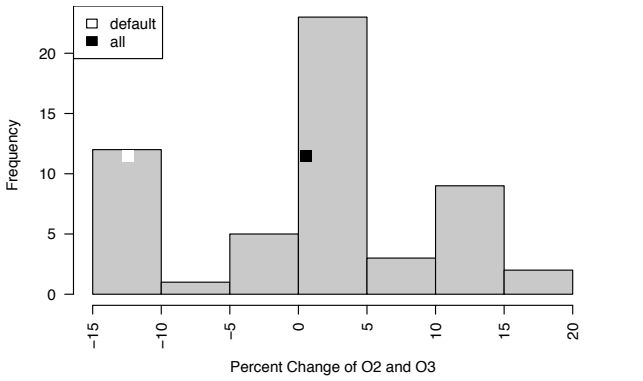

Figure 4. Quantifying libquantum's performance difference between optimization levels *O2* and *O3* on a Core2 Duo.

example, if the measurement context is biased toward $S + X$ over S for these workloads, it will inflate the benefit due to X and may even be totally misleading (e.g., X may actually degrade performance but it appears to improve performance).

This phenomenon is not specific to computer systems. In the social sciences, researchers incorporate data from subjects with diverse backgrounds so that they can factor out the effect of the subject's background on the data. For example, collecting data on work habits of graduate students at MIT and generalizing that to all students is clearly incorrect. Yet, in computer systems we routinely do this: we collect data in a single or a few measurement contexts and expect that the results hold for all measurement contexts.

3.1 How measurement context affects system behavior

Figure 3 illustrates the effect of a seemingly insignificant change in the measurement context: changing the size of the environment by a few bytes. The measurement contexts differ in the size of the value of a UNIX environment variable. A point (x,y) on the graph says that if the environment variable's value is x characters long, then the run-time of the perlbench is y% longer than if the environment variable did not exist. Each point gives the mean of 15 runs and the error bars give confidence intervals of the mean with a 95% confidence.

The results are surprising: this insignificant change can change the program's execution time by -1% to +5%. Worse, the effects are not predictable: increasing the size of the environment variable's value can improve or degrade performance! Each measurement context has its own bias. Considering that many innovations give only a few percent of improvement in performance, this bias can easily overwhelm an innovation's true behavior.

Which of the points in Figure 3 represents the true performance of perlbench? If we had measured the performance of perlbench as part of an effort to evaluate a compiler optimization, then which measurement context do we use? It may very well be that one measurement context gives the best performance for perlbench without optimizations while another measurement context gives the best performance for perlbench with optimizations. If we pick one of these extreme points, then we get skewed results: we either believe that the optimization is highly effective or highly ineffective. Thus, when evaluating an innovation it is dangerous to use just a single measurement context: chances are high that we will get at least somewhat skewed results.

Figure 4 demonstrates how ignoring the measurement context can lead to incorrect conclusions on a Core2 duo. The x-axis gives the performance change in going to optimization level *O3* from *O2* (negative numbers mean that *O3* gives a speedup) when using the gcc compiler and the SPECint 2006 *libquantum* benchmark. The y-axis gives the number of times we observed a given speedup. To generate the data in this graph we measured the benefit of each optimization level in a large number of measurement contexts. Each measurement context shifts the position of code and data in the benchmark executable file. The white "default" square gives the speedup of *O3* over *O2* when we use a single measurement context that gcc provides; this speedup is 12.5%. The black "all" square gives the speedup of *O3* over *O2* when we statistically combine the data from our many measurement contexts; this speedup is not statistically significant. In other words, if we use a single measurement context we may find that *O3* is 12.5% better than *O2*when in reality this improvement is an artifact of the measurement context.

3.2 How to avoid issues with the measurement context

Our data shows that we cannot use a single measurement context and expect to get representative results. Social scientists solve an analogous problem by collecting data from diverse subjects and then use statistical techniques to make statements about the whole population. Our proposed approach is inspired by this: we collect data from many different measurement contexts and use statistical techniques to compare distributions.

How do we generate the different measurement contexts? In our experience so far, measurement contexts often affect the performance of systems by changing how the system interacts with the many buffers (e.g., TLBs and caches) in modern microprocessors. The measurement context changes the mapping of the system to these buffers and thus changes how the system interacts with the buffers. Thus, we currently generate measurement contexts via a process we call *intervention*. We shift code and data so as to induce different interactions with the hardware buffers. We recognize that we may not cover the full space of measurement contexts using this approach but we find that our space does explore an interesting space of measurement contexts.

Figure 5 illustrates our methodology. The graph evaluates three optimization levels for mcf: the *O0-O1* points compare optimization level *O1* with *O0*, *O1-O2* points compare optimization level *O2* with *O1*, and *O2-O3* points compare optimization level *O3* with *O2*. For each pair of optimizations we have one point for one measurement context. The x-axis value gives the performance improvement obtained using the measurement context corresponding to the point. In other words, the x-axis gives the speedups we may obtain by using the prevalent methodology of using a single measurement context. The y-axis value gives the performance improve-

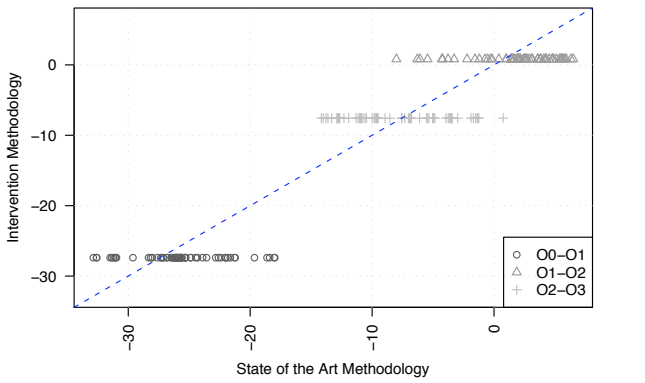

Figure 5. One versus many measurement contexts.

ment when we combine the results for all measurement contexts; i.e., our approach. Because we explore many measurement contexts, we see horizontal streaks which indicate differences between the speedup obtained from different measurement contexts. We get a single value on the y-axis for each pair of optimizations because the y-value combines information from all measurement contexts.

From Figure 5 we see that when we use only a single measurement context we can get contradictory results. For example, if we use a single measurement context to compare *O1* to *O2* we may conclude that *O2* improves performance or it degrades performance, depending on the context. Our approach combines results from many measurement contexts and thus reduces measurement context bias in our data.

4. Related Work

Korn et al. [7] evaluate the perturbation due to counter readouts by comparing the values measured on a MIPS R12000 with results from SimpleScalar's sim-outorder simulator and with analytical information based on the structure of their micro-benchmarks. Their evaluation is based on a small number of simple micro-benchmarks on a simple processor and they only consider total counts. Maxwell et al. [8] extends this work to three new platforms, POWER3, IA-64, and Pentium. This extension still focuses on micro-benchmarks and is limited to aggregate event counts. We analyze the perturbation of real benchmarks, and, in stark contrast to the above work, we find that perturbation is non-monotonic and unpredictable with respect to the amount of instrumentation.

Researchers associated with the PAPI group at the University of Tennessee at Knoxville have reported on the overhead of their publications in which they present PAPI. Moore [9] discusses accuracy issues when using PAPI for counting events. They report the overhead of starting, stopping, and reading counters in processor cycles on different architectures, but they do not study how this overhead affects measurements in real benchmarks.

In their work on flow and context sensitive profiling [1], Ammons et al. describe perturbation of hardware counters due to software instrumentation. To determine the perturbation due to their instrumentation, they compare the total metric values in a lightly instrumented system to the total metric values produced by their flow and context sensitive profiling system. In contrast, we show that one cannot generally predict perturbation in one measurement approach (e.g. a heavily instrumented system) by measuring perturbation in a different measurement approach (e.g. a lightly instrumented system).

Most fields of experimental science use inferential statistics [5] to determine whether a specific factor (e.g. drug dosage) significantly affects a response variable (e.g. patient mortality). Recent work in our field strongly highlights the absence of this practice in our area and advocates for the use of statistical rigor in software performance evaluation [3, 6].

5. Conclusions

Data collection in computer systems research is easy when compared to other sciences; we generate reams of data for each paper and it typically only takes a few days of wall-clock time. However, our collection efforts can easily produce "poor-quality data"; data that can mislead our conclusions. In this paper we show how two common measurement effects can produce poor-quality data. First, we illustrate that the observer effect can dramatically perturb our data. For example, we show that something as benign as collecting aggregate metrics with hardware support can significantly perturb our system. Second, we show that measurement context bias can significantly affect the quality of our data. For example, we show how even something as innocuous as changing the value of a UNIX environment variables can significantly affect our data.

Both of these measurement effects have solutions; *validation* to overcome the observer effect and *intervention* to overcome measurement context bias. However, these are not turn-key solutions; the validation, in particular, requires significant effort from the performance analyst.

References

[1] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[2] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the conference on Supercomputing*, 2000.

[3] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *OOPSLA'03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, pages 169–186, 2003.

[4] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, 2004.

[5] Sam Kash Kachigan. *Statistical Analysis: An Interdisciplinary Introduction to Univariate & Multivariate Methods*. Radius Press, 1986.

[6] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005)*, pages 484–490, San Diego, CA, USA, 2005. SCS.

[7] W. Korn, P. J. Teller, and G. Castillo. Just how accurate are performance counters? In *Proceedings of the IEEE International Conference on Performance, Computing, and Communications (IPCCC'01)*, pages 303–310, 2001.

[8] M. Maxwell, P. Teller, L. Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *Proceedings of the Los Alamos Computer Science Institute Symposium (LACSI'02)*, October 2002.

[9] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *Proceedings of the International Conference on Computational Science-Part II (ICCS'02)*, pages 904–912, London, UK, 2002. Springer-Verlag.

[10] Judea Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, 1st edition, 2000.