Observer Effect and Measurement Bias in Performance Analysis

Todd Mytkowicz
University of Colorado
Todd.Mytkowicz@colorado.edu

Peter F. Sweeney
IBM Research, Hawthorne
pfs@us.ibm.com

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

Amer Diwan
University of Colorado
diwan@cs.colorado.edu

University of Colorado at Boulder
Technical Report CU-CS 1042-08
June 2008

Observer Effect and Measurement Bias in Performance Analysis

Todd Mytkowicz
University of Colorado
Todd.Mytkowicz@colorado.edu

Peter F. Sweeney
IBM Research, Hawthorne
pfs@us.ibm.com

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

Amer Diwan
University of Colorado
diwan@cs.colorado.edu

Abstract

To evaluate an innovation in computer systems, performance analysts measure execution time or other metrics using one or more standard workloads. The performance analyst may carefully minimize the amount of measurement instrumentation, control the environment in which measurement takes place, and repeat each measurement multiple times. Finally, the performance analyst may use statistical techniques to characterize the data.

Unfortunately, even with such a responsible approach, the collected data may be misleading. This paper shows how easy it is to produce poor (and thus misleading) data for computer systems due to observer effect and measurement bias. Observer effect occurs if data collection perturbs the behavior of the system. Measurement bias occurs when a particular environment in which the measurement takes place favors some configurations over others. This paper demonstrates that observer effect and measurement bias have significant impact on performance and can lead to incorrect conclusions. These effects are large enough to easily mislead a performance analyst. Nevertheless, in our literature survey of recent PACT, CGO, and PLDI papers we found that papers rarely acknowledged or used reliable techniques to avoid observer effect or measurement bias.

We describe and demonstrate techniques that help a performance analyst identify situations when they have poor quality data. These techniques are based on causality analysis and statistics which natural and social sciences routinely use to avoid the observer effect and measurement bias.

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 minimize the amount of measurement instrumentation, attempt to control everything in the environment that can influence performance (the *measurement context*), and repeat each measurement multiple times. Finally, the performance analyst may use statistical techniques to characterize the data.

Unfortunately, even with such a responsible approach, the collected data may be misleading. This paper shows how easy it is to produce poor (and thus misleading) data for computer systems research, explores two common causes of poor-quality data, and discusses two techniques that can help to identify poor-quality data.

What is poor-quality data? We define poor quality data as data that may cause us to draw invalid conclusions. High quality data, on the other hand, allows us to draw valid conclusions about the system that we are evaluating. High-quality data does not mean that the data is perfect: it just means that it was good enough *for our needs*. Thus, high-quality is not an absolute characterization of the data; instead it depends on the particular task for which one is using the data. In this paper, we explore two common causes of poor-quality data.

First, we may get poor-quality data if our data collection alters the performance of the system being measured; the scientific literature calls this the *observer effect*. We show that even a seemingly insignificant measurement probe can dramatically alter system behavior. We found in our literature survey of recent PACT, PLDI, and CGO papers, that it is rare (1 out of 72 papers) to find a paper that acknowledges and uses reliable techniques for avoiding the observer effect.

Second, we may get poor-quality data if we compare two systems (or two variants of the same system) and the measurement context favors one system (or variant) over the other; the scientific literature calls this the *measurement bias*. We show that different measurement context (e.g., settings of environment variables) favor different systems (or variants) differently. Thus, the measurement context can show a particular system or variant in an optimistic or a pessimistic light; we cannot tell which one it will be in advance. We found in our literature survey that none of the 72 papers reviewed acknowledge or use reliable techniques for avoiding the type of measurement bias we describe in this paper.

The implications of observer effect and measurement bias are profound for experimental work in computer systems. The observer effect can mislead a performance analyst about the true cause of a system's poor performance: as a hypothetical example, the data may suggest that the system has poor performance due to L1 data cache misses but in reality the measurement probes (directly or indirectly) caused the cache misses. The measurement bias can lead to misleading conclusions about the benefit of a new idea. For example, consider a performance analyst who wants to determine if idea, I, is beneficial for system, S. If the performance analyst measures S and $S + I$ in a measurement context that favors $S+I$, she may conclude that I is beneficial even when it is not. We show that the above examples are not rare hypothetical scenarios; instead both the observer effect and measurement bias are frequent and large enough to cause a performance analyst to draw invalid conclusions.

We do not claim that the measurement bias and observer effect are the only causes of poor-quality data; there may be other causes as well. In this paper, we focus on the measurement bias and observer effect, because, even though they seem obvious in hindsight, they are largely ignored by systems researchers.

Our literature survey indicates that the primary approach in prior work to eliminate observer effect is to run experiments on a lightly-loaded system while using minimal instrumentation. We show that even minimal instrumentation (a few added instructions) can cause a significant observer effect. To avoid the measurement bias, most researchers use, not a single workload, but a set of workloads (e.g., all programs from a SPEC benchmark suite) in the hope that the bias will statistically cancel out. For this to work, we need large and diverse set of workloads. Unfortunately, most benchmark suites have biases of their own and thus will not cancel out the effects of measurement bias; e.g., the DaCapo group found that the memory behavior of the SPECjvm98 benchmarks was not representative of typical Java applications [4].

In this paper, we discuss two techniques that can help detect observer effect and measurement bias. First, we propose and demonstrate the use of causal analysis [18] to detect misleading data. The basic idea is to use intervention to test the conclusions that we have drawn from our data. For example, if our data indicates that the poor performance of our system is due to a particular data structure not fitting in the cache, we need to find a way to make the data structure fit in the cache (the intervention) and confirm that the poor performance does actually go away. In this way we gain confidence that our conclusions are valid and not an artifact of poor-quality data. This approach is labor intensive but that is unavoidable in general. We demonstrate (Section 5.1) this approach using a realistic example from our own research.

Second, we also describe *variant generation*, an approach that is not as labor intensive as causal analysis but is also less powerful. The basic idea is to generate many measurement contexts, collect data in all the contexts, and reason with the data using standard statistical techniques. The effectiveness of this approach depends on how thoroughly we are able to explore the space of measurement contexts; if the generation of measurement contexts is itself biased then this approach is ineffective. Variant generation helps only with the measurement bias and is ineffective against the observer effect.

The remainder of the paper is structured as follows. Section 2 reviews the state-of-the-art of performance measurement in systems research. Section 3 presents our experimental methodology. Section 4 studies observer effect and measurement bias, and shows how easy it is to get "poor-quality" data. Section 5 suggests ways of detecting with them. Section 6 compares this paper to related work, and Section 7 concludes.

2. Literature Review

When we show our data to systems researchers, their first reaction is: "This was all known"[1]. In this section we show that even if these phenomena are well known to systems researchers, almost none of the papers in 2007 CGO [1], PACT [16], and PLDI [17] explicitly address the observer effect (Section 4.1) and measurement bias (Section 4.2). Thus, while systems researchers may be knowledgeable about the phenomena, perhaps they do not realize how severe they are.

[1] This is a quote from a review of an earlier version of this paper.

We picked CGO, PACT, and PLDI for our literature survey because they are all highly selective outlets for experimental computer science work. In fact, of the 102 papers in these three conferences, 72 had at least one section dedicated to experimental methodology and evaluation. The remainder of this section focuses on these 72 papers. When in doubt, we always gave the benefit of the doubt to a paper's methodology.

2.1 Papers that use simulations

Many researchers use simulations since they enable them to try out hypothetical architectures. Simulations can also avoid the observer effect, though as we show in Section 4.3, they do not help with measurement bias. 25 of the 72 papers we reviewed used simulations. Unfortunately, simulations themselves can be inaccurate [5]. Thus using simulations trades off observer effect for simulator inaccuracy.

2.2 Papers that report speedups

If the ideas in a paper offer huge speedup then it can be argued that the speedup is immune to observer effect and measurement bias. For example, if a technique gives a many-fold speedup for real applications, it is unlikely that the speedup is due to the observer effect. Of the 72 papers we surveyed, 53 presented speedups. The mean speedup reported by these papers was $9.3\% \pm 2.6\%$ which is small enough that it can easily be overwhelmed by the observer effect or measurement bias.

2.3 Papers that acknowledge the observer effect

Of the 72 papers we reviewed, 46 papers reported data for real hardware (i.e., not simulations) and thus these 46 papers are amenable to the observer effect. 14 of those papers mentioned the amount of overhead their measurement infrastructure incurred (usually a few percentage increase in run time). As we show in this paper, a minimal increase in runtime can drastically alter benchmark performance. Of the 14 that mention overhead, only one paper[11] actually did anything to *understand* the complexities of their measurement infrastructure and how it influenced collected metrics.

2.4 Papers that acknowledge measurement bias

Unlike the observer effect, measurement bias affects data collected on real hardware and on simulators (Section 4.3). In this paper, we focus on two specific instances of measurement bias (shell environment state and linking order of the .o files of a benchmark) and demonstrate that they can cause poor-quality data and result in invalid conclusions.

Although none of the papers we reviewed said anything about bias due to environment variables or linking order, most (65) papers used more than one benchmark or input sets for their evaluation, which partially solves the measurement bias. If we use a sufficiently large and diverse set of workloads along with careful statistical methods, most measurement bias should get factored out. Unfortunately, there is no reason to believe that our benchmark suites are diverse; indeed there is some reason to believe that they themselves are biased. For example, the designers of the DaCapo benchmark suite found that the commonly used benchmark suite, SPECjvm98 was

Benchmark	Description	Cycles × 1e+09
perlbench	Scripting language interpreter	1.5
bzip	Compression algorithm	49
gcc	C compiler	1.4
mcf	Single-depot vehicle scheduler	210
gobmk	Go program	400
hmmer	Computational biology DNA sequence search	390
libquantum	Quantum computer simulator	30
h264ref	Video encoding	110
sjeng	Chess program	910

Table 1. Benchmarks used in our experiments.

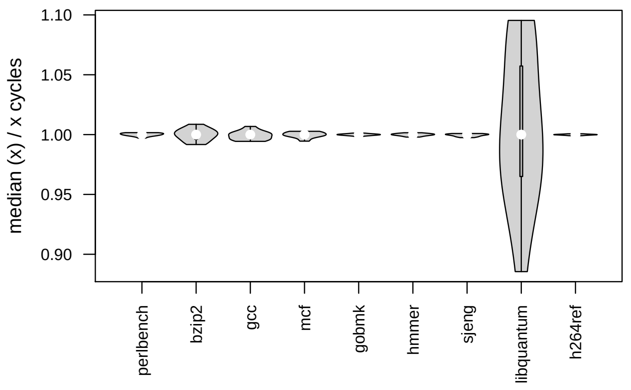

Figure 1. Variation of execution time (in cycles) across runs of the SPECint C 2006 benchmarks.

unrepresentative of real workloads in terms of their memory usage pattern [4].

2.5 Summary

In summary, while researchers might know about observer effect and measurement bias, we were unable to find any paper that used a comprehensive strategy for avoiding these issues.

3. Experimental Methodology

We now describe our benchmark programs, hardware infrastructure, and approach to measurement

3.1 Benchmarks

We use the SPECint 2006 [20] benchmarks (**Table 1**) with the *train* inputs for our experiments. All benchmarks were compiled with gcc version 4.2.1 and optimization level *O2*.

Figure 1 shows inter-run variation for all our benchmark programs: each violin plot [7] shows the variation of a single benchmark between identical runs of the benchmark (15 runs per benchmark in total). The y-axis gives the ratio of a given the median time to a particular run time. The white dot in each violin plot gives the data for the median run (and thus the ratio is always 1) and the thick line through the white dot gives the inter-quartile range. The width of the violin plot at y-value y is proportional to the number of runs with ratio y. From this graph we see that all benchmarks with the exception of libquantum have tight inter-run variation: in other words, the run time of the benchmark varies little from run to run.

This graph gives a baseline for all other graphs in this paper; for example, if the difference between the performance of a program compiled with *O2* and compiled with *O1* is smaller or comparable to the inter-run variation then we cannot credit the difference to the difference between *O2* and *O1*. The inter-run-variation quantifies *random error*. Statistical rigor (performing multiple measurements, computing the mean, and using the confidence intervals for the mean) addresses this error. The remainder of this paper is concerned with *systematic error*, a form of error that no amount of statistical rigor can overcome.

Parameter	Dual-Core Xeon 5160	Pentium 4
Operating System	Linux 2.6.17	Linux 2.4.21
Tool Chain	gcc 4.1.1	gcc 4.2.1
Micro-architecture	Core	NetBurst
Clock Frequency	3 GHz	2.4 GHz
memory	8G	2G
L1	32K Ins., 32K Data	12K Ins. 8K Data
L2	128K Unified	512K Unified
L3	4096K	NA
TLB entries	512	64

Table 2. Machines used to collect data.

3.2 Hardware infrastructure

We conducted our experiments on two machines: a Pentium 4 and a Dual-Core Xeon (**Table 2**). Unless we state otherwise, all data in this paper is for the Pentium 4 workstation; we picked the Pentium 4 for the majority of our experiments since the Pentium 4 has the best support for collecting performance data (hardware performance monitors) of any recent computer. For example, the Pentium-4 can collect up to 18 hardware metrics at a time; in contrast the Core Duo can only collect 2 at a time. To confirm that the phenomena explored in this paper are not specific to the Pentium 4, we will also present results from the Dual-Core Xeon and from the m5 simulator using the O3CPU model [3].

3.3 Our approach to measurement

With all aspects of our measurements we attempted to be as careful and diligent as possible. In other words, the observer effects and measurement bias that we demonstrate later in the paper are present despite our best efforts.

- Except in the experiments where we add environment variables, we conducted our experiments in a minimal environment (i.e., we unset all environment variables that were inessential).
- We conducted all our experiments on minimally-loaded machines, used only local disks, and repeated each experiment multiple times to ensure that our data was representative and repeatable.
- We conducted our experiments on two different sets of hardware and (when possible) one simulator. This way we ensured that our data was not an artifact of the particular machine that we were using.
- Some Linux kernels (e.g., on our Dual-Core Xeon) randomize the starting address of the stack (for security purposes). This feature can make experiments hard to repeat and thus we disabled it for our experiments.
- Except in the case of software instrumentation, we dynamically added all our instrumentation in a wrapper around the main function. The wrapper programs what hardware metrics to collect before main executes and reads the hardware metrics out after main executes. We use the "LD_PRELOAD" feature to install our wrapper. With this

approach, our instrumentation did not directly interact with the compilation of the benchmark program.

4. Problems

We show that observer effect (Section 4.1) and measurement bias (Section 4.2) are significant problems for performance analysts.

4.1 Observer Effect

Performance analysts often prefer to collect data using hardware performance monitors because it is cheap in terms of instructions added to the measured software: the hardware increments the counters and the software only needs to initialize and read out the counter values. However, hardware monitors cannot collect some kinds of data (e.g., number of database transactions) and thus, performance analysts sometimes also collect software metrics. Unlike hardware metrics, software metrics insert instructions into the code to increment software monitors and when executed the increments increase the number of instruction executed [2].

Precisely quantifying the observer effect is difficult if not impossible. If we had data from a pristine run (i.e., with no observer effect) then we could compare a pristine run to other runs to quantify the observer effect. However, we have no way of getting a pristine run: even when we collect only hardware metrics we may still suffer from the observer effect. The approach we take is to vary the amount of instrumentation (for hardware or software metrics) and see how the performance of our benchmark changes. If we see that the performance is very different between instrumentation I and instrumentation J then we know that at least one of them must have suffered from the observer effect.

4.1.1 Observer effect when collecting only hardware metrics

To understand the performance of an application, a performance analyst can collect a number of hardware metrics using the hardware performance monitor facility in the hardware. For example, the performance analyst may collect not just total cycles but also other metrics such as I- and D-cache misses. This section shows that this data collection may alter the data; specifically the performance analyst may end up with a different number of total cycles when she collects total cycles by itself than if she collects it with other metrics.

There are many different ways in which we can record the values of hardware metrics. The most obvious approach is to add instrumentation to the program's source (either manually or using an instrumentation tool) that invokes the appropriate operations for accessing the hardware performance monitors. We decided against this approach because we feared that the instrumentation may cause the compiler to produce different code (e.g., by affecting register pressure). Instead, we use a precompiled wrapper around the `main` function; we use environment variables to communicate the set of metrics to collect to the wrapper. In this way, we avoid having to recompile either the program or the wrapper when we need to collect a new set of hardware metrics. This is a common approach for collecting data from HPM's without altering the source code of the application. We modeled our methods after a number of tools used in the field (IBM's hpmcount[3], and the University of Oregon's TAU[4] to name a few). In Section 5.1, we will show that the use of environment variables is the cause of the perturbation.

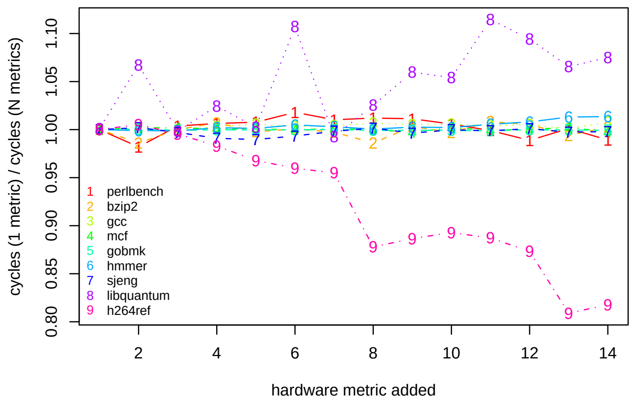

Figure 2. The effect on a benchmark's performance (total cycles) as we collect additional hardware metrics.

Figure 2 shows how the number of hardware metrics collected in a program run affects measurement results (in this case, the overall cycle count). The x-axis shows the number of hardware metrics to be collected in a run. The x-axis starts at x=1 (the baseline) because we have to collect at least one metric (the cycle count). The y-axis shows the speedup over the baseline run. Speedup is computed as the baseline run's cycles divided by a given run's cycles. That is, if a given run incurs fewer cycles than the baseline, the speedup is less than one. The figure contains one curve for each of the nine benchmarks. To factor out random error we performed five measurement runs for each benchmark and metric count. Each point on each curve corresponds to the mean speedup over these five runs. We computed the 95% confidence intervals for all points and with the exception of libquantum the intervals were very tight; thus we omit them from the graph to avoid visual clutter. Even though libquantum has a wide confidence interval due to its large inter-run variation (Figure 2), we found that the variations for libquantum (and all other benchmarks) in Figure 2 are statistically significant.

Figure 2 shows that adding hardware performance monitors can significantly change total cycles. The most extreme cases are libquantum and h264 whose highest and lowest points differ by 0.12 and 0.20 respectively. That is, even just the act of collecting a few hardware metrics can change overall execution time by a factor of up to 0.20 for our benchmarks. Thus a program may appear to be much faster or slower; as a consequence a performance analyst may either waste time trying to optimize a program that does not need optimization or conversely not optimize a program even when it needs optimization. Observe that these changes in performance are often much larger than the inter-run variation of the benchmarks (Figure 1). While rigorous computer system performance analysis approaches advocate taking inter-run variation into consideration[6], we are not aware of approaches that also address the observer effect we show here.

We also note the lack of any obvious trend in Figure 2. We would expect that more instrumentation always leads to a higher cycle count but this is clearly not the case. Instead

[2] To minimize the number of additional executed instructions, we do not insert a conditional around increments.

[3] http://publib.boulder.ibm.com/infocenter/systems/index.jsp?topic=/com.ibm.aix.cmds/doc/aixcmds2/hpmcount.htm.

[4] http://www.cs.uoregon.edu/research/tau/home.php.

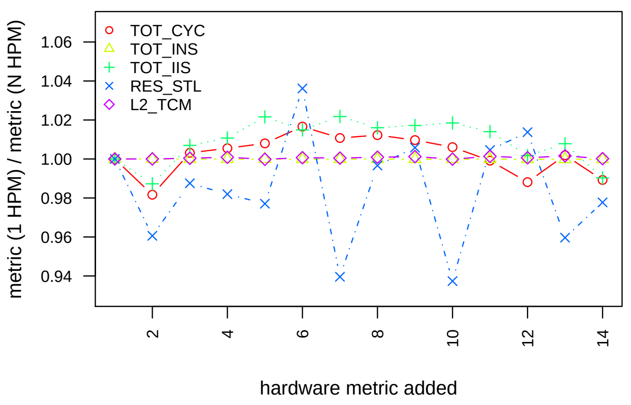

Figure 3. The effect on a hardware metric's value for perlbench as we collect additional hardware metrics.

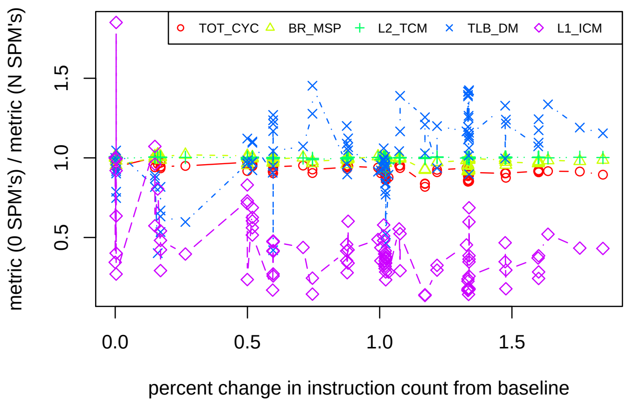

Figure 4. The effect on a hardware metric's value for perlbench as we collect additional software metrics.

the cycle count changes unpredictably with the amount of instrumentation. Besides libquantum and h264ref, which exhibit extreme sensitivity to the number of hardware metrics (e.g. for libquantum, measuring two counters instead of one leads to a speedup of 1.07), two other benchmarks (perlbench and bzip2) also show sensitivity (speedup varies between 0.98 and 1.01 in both cases).

To shed further light onto the observer effect due to collecting only hardware metrics, **Figure 3** plots the change in values for five frequently occurring hardware metrics for perlbench. This figure has the same structure as Figure 2, with the exception that the different curves represent different metrics instead of different benchmarks. We present data only for metrics that occur frequently because a change in an infrequent metric is often not interesting; e.g., going from 1 to 2 TLB misses doubles the number of TLB misses but it is not an interesting change. We present data for perlbench because for the most part, its graphs visually look "average" compared to the other benchmarks; we could not include the full set of graphs due to space considerations.

From Figure 3 we see that adding one hardware metric can significantly affect the values of other metrics. For example, measuring additional hardware metrics causes *RES_STL* (cycles stalled on any resource) to vary between 0.94 and 1.04 times its value when it is collected on its own. Such swings can easily mislead a performance analyst. For example, this observer effect may cause a performance analyst to waste time tracking down the cause of stalls when the stalls are insignificant in the pristine run.

4.1.2 Observer effect when also collecting software metrics

Let's suppose a performance analyst wants to study the relationship between the number of heap allocations and the number of data cache misses. The analyst instruments the heap allocator to update a software performance counter (a global variable) that counts the number of allocations. This section shows that collecting the software metric (e.g., heap allocations) can alter the value of the hardware metric (e.g., data cache misses).

Unlike hardware metrics, software metrics need instrumentation added to a program to increment counters in addition to initializing and reading them out. The measurement infrastructure we use in this section is based on CIL [15] and can selectively add instrumentation to a program. Using profiling runs, we identified key events (mainly calls) in each of our benchmark programs; these are the events that a performance analyst would probably measure if she wanted to understand these programs. Our infrastructure enables us to add instrumentation for any subset of these events. The instrumentation for an event takes the form of an increment to a global variable. Collecting a different number of software metrics requires re-running CIL to instrument the program and then recompiling the instrumented program. In all experiments in this section we collect eight hardware metrics in addition to the zero or more software metrics.

Figure 4 illustrates how the number of software metrics collected in perlbench affects the measurement count of hardware metrics. As we move along the x-axis we add more and more software instrumentation to perlbench, starting from zero SPMs (the baseline) to the number of SPMs that causes a 1.8% increase in instructions executed. Figure 4 has one curve for each of the five most frequently occurring hardware metrics. The y-axis represents the speedup over the baseline (i.e., no software metrics). Figure 4 shows that (i) adding an insignificant number of instructions can have a great impact on the performance of a system (i.e., large observer effect); and (ii) the effect is not predictable in that adding instrumentation can improve or degrade the values of the hardware metrics.

Figure 5 shows the effect of collecting software metrics on total cycles for all of our benchmark programs. The y-axis gives the ratio of execution time when we collect no software metrics to the execution time when we collect some software metrics. Each violin plot visualizes the distribution of speedup as a result of collecting different numbers of software metrics. Each point that contributes to a violin plot collects a particular set of software metrics; different points collect different sets of metrics. The left-most violin plot (perlench) summarizes the points in Figure 4 for total cycles (TOT_CYC). The white dot in each violin plot gives the median speedup, and thick lines through the white dot give the inter-quartile range. The width of the violin plot at a particular y value gives the proportion of metric sets that lead to a speedup of y. The number below each violin plot is the maximum percentage change in the number of executed instructions as a result of the instrumentation.

From Figures 5 we see that adding a small amount of software instrumentation can dramatically change the execution time of the program. For example, by changing the

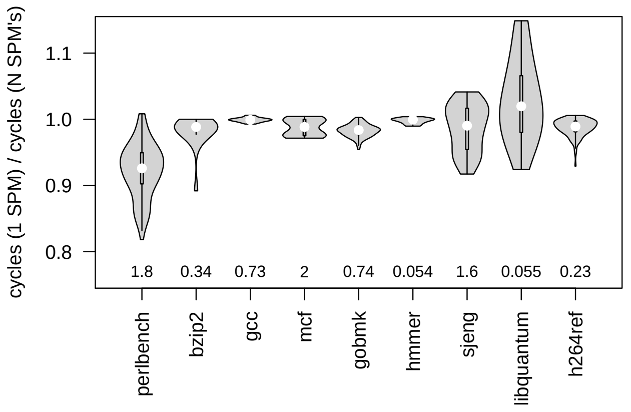

Figure 5. The effect on a benchmark's performance (total cycles) as we collect additional software metrics.

instruction count of perlbench by less than 1.8% we changed perlbench's performance drastically, ranging from a speedup of 1.15 to a slowdown of 0.92. This is unexpected: we expect that such a small change in instruction count would not have such a large impact, but it does!

Our results show that even if the overhead due to software metrics is small (e.g. only 2% increase in dynamic instructions), the observer effect may be enormous. For example, the hapless performance analyst may spend all of his time tracking down the source of I-cache misses in perlbench, without realizing that software instrumentation caused more than 80% of those misses.

4.1.3 Summary of observer effect

We find that the observer effect is prominent even when the observations themselves are lightweight. Worse, this effect is not predictable: more observations do not translate into more observer effect. Thus, unlike the approach taken in prior work [19] we cannot just subtract out the observer effect after collecting our data.

4.2 Measurement Bias

Systems researchers often need to compare two versions of a system; for example compiler researchers often need to compare a system with and without some compiler optimization. If the measurement context favors one of these versions over the other then the comparison will be unfair: in our example, it may either exaggerate or play down the benefit of the optimization.

This section shows that measurement bias is significant; e.g., with some contexts, gcc's optimization level *O2* outperforms *O3* and with other contexts *O3* outperforms *O2*. In this section, we explore two kinds of changes between measurement contexts: settings of environment variables and linking order. The goal of this section is not to enumerate all possible sources of bias but to show that bias is large enough to mislead the performance analyst.

4.2.1 Measurement Bias due to Shell Environment State

Figure 6 shows the effect of adding environment variables to speedup obtained with "gcc -O3" versus "gcc -O2". The leftmost point is for an environment of 2 bytes; all subsequent points add 390 bytes to the environment; the last point is at 8192 bytes. To add to an environment, we simply extend the

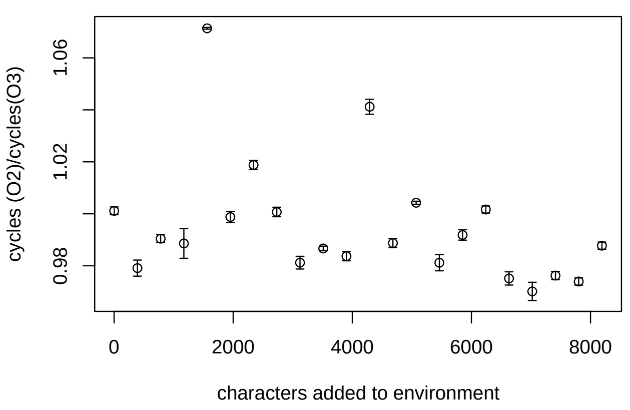

Figure 6. The effect of adding environment variables on the speedup of *O3* over *O2* for Perlbench.

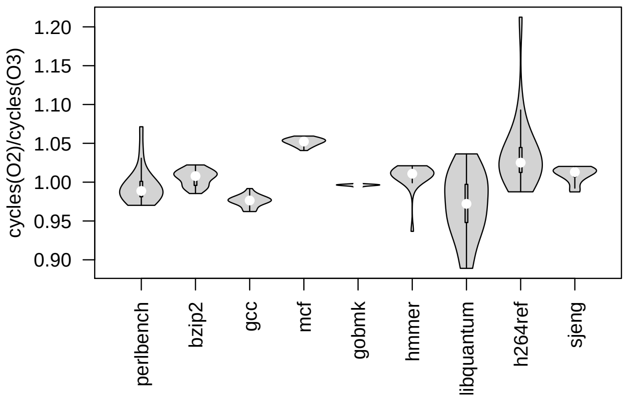

Figure 7. The effect of adding environment variables on speedup of *O3* over *O2* for our benchmarks.

string value of a dummy environment variable that is not used. Thus, the only affect of this addition is to affect the starting address of the stack (the environment variables occur before the call stack in memory).

A point (x, y) says that when we have x bytes in the environment, the execution time with *O2* divided by the execution time with *O3* is y. We generated this data using the $bash$ shell[5]. We computed each point using fifteen runs each with *O2* and *O3*; the error bars give the 95% confidence intervals around the mean. The tight confidence intervals means that our runs are easily reproducible. We repeated these experiments with other commonly-used shells and obtained similar results.

The most important point to take away from this graph is that depending on the environment variables we may conclude that *O3* is better than *O2* (i.e., the y value is less than 1.0) or *O2* is better than *O3* (i.e., the y value is greater than 1.0)! As discussed above, we repeated the experiment for each data point multiple times; thus the extreme points are not anomalies but true reproducible behavior.

Figure 7 summarizes similar data across all benchmarks. Each violin plot gives the data for one benchmark and plots all the points for the benchmark (each point corresponds to

[5] http://www.gnu.org/software/bash/manual/bashref.html

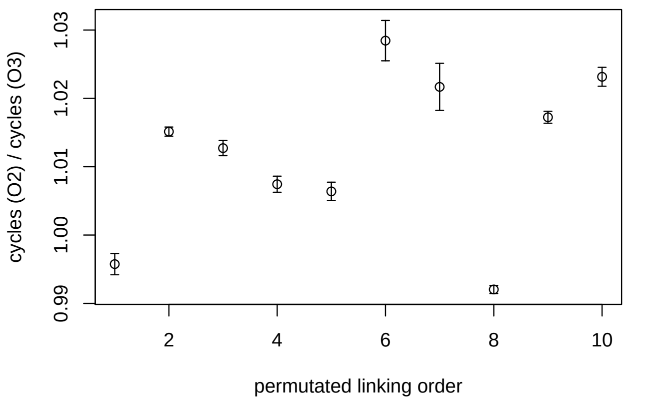

Figure 8. The effect of linking order on speedup of *O3* over *O2* for the perlbench benchmark.

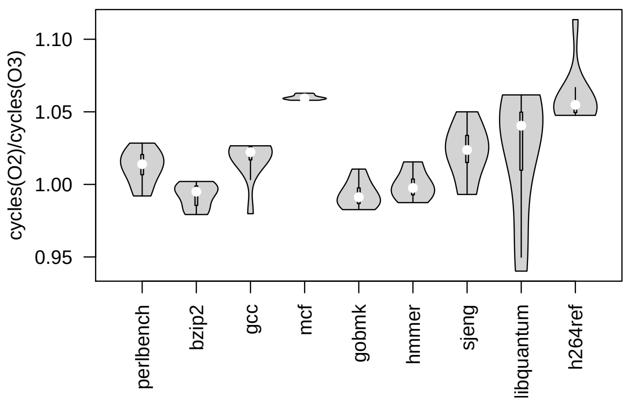

Figure 9. The effect of linking order on speedup of *O3* over *O2* for the SPECint benchmarks.

a particular size of environment). The left-most violin plot summarizes all the data presented for perlbench in Figure 6. We see that six of the nine violin plots straddle 1.00: this means that depending on the environment, *O3* either offers a speedup or a slow down for all the benchmarks.

In Figure 7, the difference between the maximum and minimum points of the violin plot are particularly instructive since they give an indication of how much of a bias one can end up with. The most extreme is h264ref which varies from 0.99 (i.e., slight slow down due to *O3* over *O2*) to 1.21 (i.e., a healthy speedup of *O3* over *O2*). This variation is large enough to easily eclipse the speedup due to many compiler optimizations [12].

4.2.2 Measurement Bias due to Linking Order

Figure 8 explores the effect of object linking order on the speedup of "gcc -*O3*" over "gcc -O2". To obtain this data, we compiled perlbench 10 times, each time randomly changing the order of the *.o* files to the linker. A point (x, y) in Figure 8 says that for the x^{th} linking order we tried (we generated these randomly) the execution time of the benchmark with *O2* over its execution time with *O3* was y. For each point, we conducted 15 run each with *O2* and *O3*; the whiskers give the 95% confidence intervals around the mean.

The most important point to take away from Figure 8 is that depending on the linking order *O3* can either give a speedup over *O2* (i.e., y value is less than 1.0) or a slow down over *O2* (i.e., y value is greater than 1.0).

Figure 9 uses a violin plot to illustrate the effect of linking order on the speedup of *O3* over *O2* for all the benchmarks. Each violin plot summarizes data for all the linking orders for the benchmark. The left-most violin plot summarized the data plotted for perlbench in Figure 8. From this figure we see that seven of the nine violin plots straddle 1.0; thus, for these benchmarks, we may arrive at completely conflicting conclusions about the benefit of *O3* over *O2* depending on the linking order that we use.

4.2.3 Summary of Measurement Bias

We have shown that measurement bias can easily mislead a performance analyst into believing that one configuration is better than another whereas if the performance analyst had conducted the experiments in a slightly different measurement context she would have concluded the exact opposite. Moreover, the effects of measurement bias are large: they can easily overwhelm the performance benefits due to most optimizations. Even worse, measurement bias appears to be unpredictable: for example, adding environment variables can increase or decrease the speedup due to *O3*.

Comparing Figure 9 to Figure 7 we see that adding environment variables typically has a larger effect than changing linking order. Nonetheless, both are significant sources of measurement bias in their own right.

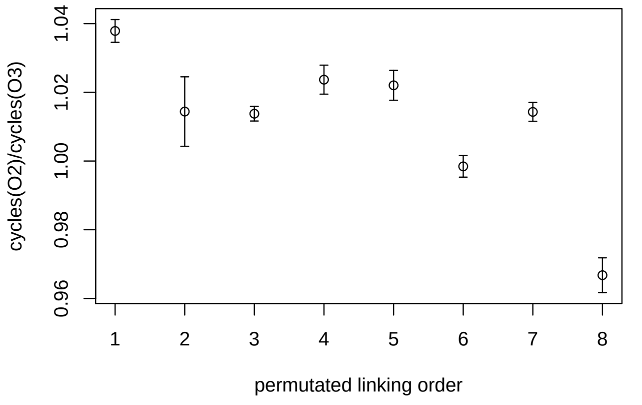

Figure 10. The effect of linking order on speedup of O3 over O2 for the perlbench benchmark on Core Duo workstation.

4.3 Generality of our results

Figure 10 presents data similar to Figure 8 except for the Core Duo workstation. We see that our phenomenon is not specific to the Pentium 4: indeed even on the Core Duo we see that the linking order can lead to conflicting conclusions over the benefit of optimization level O3 over O2. Furthermore, the range of values for different linking orders is greater on the Core Duo, from 0.96 to 1.07, than the range on the Pentium 4, from 0.99 to 1.03.

Figure 11 is similar to Figure 8 except it presents data for the m5 simulator (o3cpu model) and for the bzip2 benchmark running a small (home grown) input. We use bzip2 with a small input because the larger inputs or other benchmarks were prohibitively slow. We see that the measurement bias also affects simulation! Thus, while simulations may be a reasonable approach for avoiding the observer effect, they do not help with measurement bias.

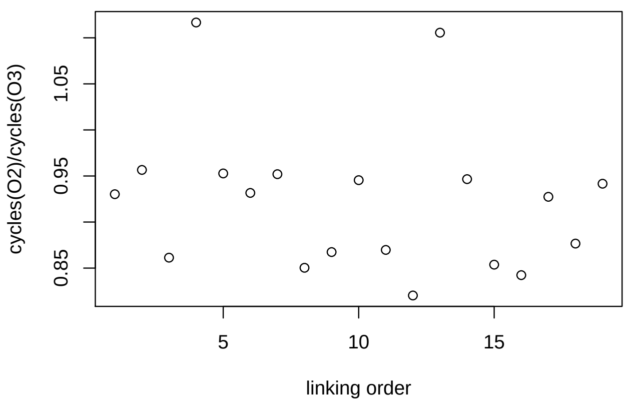

Figure 11. The effect of linking order on speedup of O3 over O2 for the bzip2 benchmark on m5 simulator.

5. Recognizing Poor-Quality Data

So far we have given just the bad news: that even with a careful experimental methodology we can end up with a significant observer effect and measurement bias. Now, we present two approaches that attempt to recognize poor-quality data and thus prevent one from drawing invalid conclusions.

5.1 Using causal analysis

We propose using casual analysis [18] to determine if poor-quality data is misleading us. More concretely, let's suppose that our data leads us to the conclusion that X causes Y (e.g., X may be L1 D-cache misses and Y may be instructions-per-cycle). In reality, it may be that X does not actually cause Y; instead, the observer effect has led us to the wrong conclusion. To check if this has indeed happened, we use intervention: we change X while attempting to keep everything else constant and see if Y changes accordingly. If Y changes accordingly, then we have some evidence that our conclusion is true and not caused by poor-quality data. The remainder of this section demonstrates such causal analysis on a concrete case study.

Figure 2 (in Section 4.1) showed that adding hardware performance monitors could lead to significant observer effect. In the terminology of causality, we concluded the following from our data: the act of collecting hardware metrics caused a change in the execution time of the program. We now explore this causal relationship.

As discussed in Section 4.1, we were careful to avoid recompilation; in other words, we executed the same code for the program when collecting n metrics and collecting m metrics (obviously, the instrumentation code behaved differently for the different numbers of metrics). Thus, the only difference between n and m metrics were: (i) the dynamic instructions to set up and read out the different number of metrics; (ii) the dynamic instructions to interpret the environment variables which we used to communicate the set of metrics that we wanted to collect; and (iii) the memory layout changes we incurred due to a different number of environment variables that are used to specify what hardware metrics to collect.

Because Section 4.2.1 had already warned us that the settings of environment variables could cause significant measurement bias, we decided to explore (iii) first. Specifically, we changed our measurement infrastructure so that we used exactly the same number of environment variables with the same length for any number of hardware metrics; effectively we padded the variables to be of the same size.

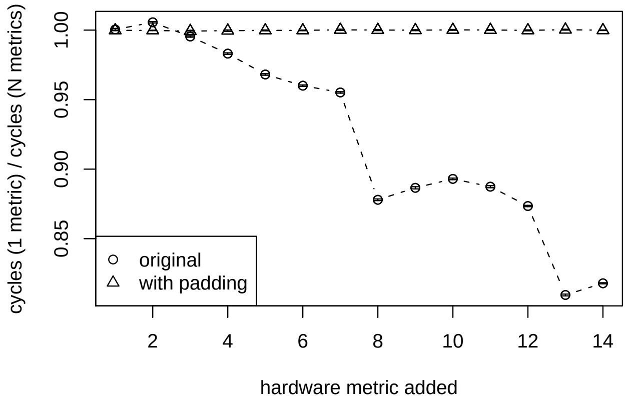

Figure 12. Change in h264ref's total cycles as we add hardware metrics.

Figure 12 illustrates the how the total cycles changes as we add more metrics for the h264ref benchmark. The "original" curve gives the data using the approach in Section 4.1; the "with padding" uses padding to keep the environment at a fixed size. A point (x, y) says that when we collect x hardware metrics, the ratio of the cycle count when collecting one metric and when collecting x metrics is y.

From Figure 12 we see that adding hardware metric has a significant impact on the "original" approach: indeed this is the variation that accounts for the measurement bias in Section 4.2.1. On the other hand, from the "with padding" graph we see that as soon as we fix the size of the environment variables, the variability goes away: the curve is essentially flat! In other words, we have proved that it was not the collection of hardware metrics that was affecting total cycles but the way in which we were using environment variables.

While the above analysis shed light on whether or not a particular conclusion inspired by our data was true, it does not say anything about the data in an absolute sense of the word. Even with the padding intervention we did not eliminate the observer effect: we simply reduced it to the point where it showed that our initial conclusion was false. Indeed, except in very limited cases, one cannot eliminate the observer effect; we simply have to learn to work around it.

Causal analysis is not a turn-key solution. It requires careful judgment on the part of the performance analyst. Even in our demonstration, we could have followed many different paths. For example, instead of padding environment variables, we could have added environment variables for the different number of metrics while always collecting only a single metric. This analysis too would have led us to the same conclusions.

In summary, this section demonstrated how to use causal analysis to test a conclusion inspired by the data. Our demonstration actually showed that a causal relationship that seemed obvious was actually *not* a causal relationship! We have repeated such analysis many times for different scenarios. Sometimes we find that the causal analysis refutes our conclusion and in other cases it gives us further confidence that our conclusion is correct.

5.2 Using variant generation

Causal analysis provides the most reliable way of determining whether or not measurement bias and observer effect

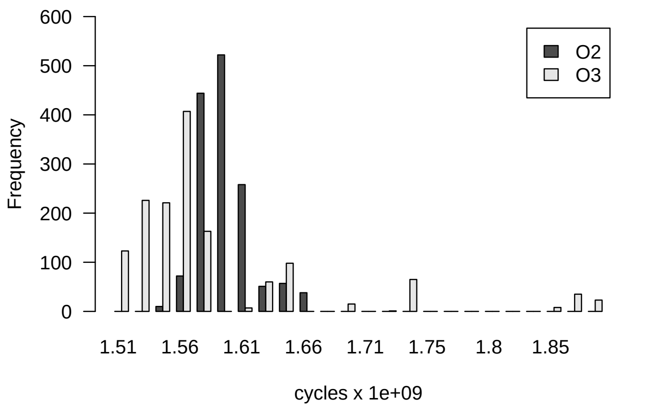

Figure 13. Using variant generation to determine the speedup of *O3* over *O2*. At a 95% confidence interval, we estimate the speedup to be 1.007 ± 0.003

have led us to an incorrect conclusion. However, as seen in the previous section, causal analysis requires a significant time-investment from the researcher: rewriting part of the measurement infrastructure. In this section, we describe an automated approach that *potentially* relieves the amount of effort required by a researcher. This new approach only detects measurement bias, but not observation effect.

In Section 4.2, we identified two sources of bias in our measurements (shell environment state and the linking order of a benchmark's object files) and demonstrated their effect when we evaluate the speedup of *O3* over *O2*. Given our prior analysis, a question still remains: "What is the speedup of *O3* over *O2*"? Although we cannot answer this question in general because we cannot enumerate all possible sources of measurement bias, we can at least attempt to answer it for the source of measurement bias that we know about.

The basic idea is as follows: rather than using a single measurement context to collect our data, we use interventions to generate a large number of contexts. We can generate the contexts by varying parts of the measurement context we know to cause measurement bias; for example, we can try many different linking orders and many different settings and sizes for environment variables. If we randomly and adequately sample from all possible contexts we have factored out the effects of these two possible causes of measurement bias; effectively we have turned the systematic bias introduced by shell environment state and linking order into random error and thus with rigorous statistical techniques we can factor out this random error.

In this section, our goal is to understand the performance benefit of *O3* over *O2*, and after a large collection effort (mostly just machine time) we can compare the two distributions to see if there is any statistically significant difference in the means. To do this, we use the paired t-test[8], as we make sure that run 10 from *O3* is collected in the same measurement context as run 10 from *O2*.

Figure 13 demonstrates our approach by plotting the distribution of 484 measurement contexts (using a cross product of 22 permutations of linking order and 22 permutations of shell environment state) for *O2* and for *O3*. We capture 3 runs per linking order and shell environment state to mitigate the effects of inter-run variation. Our hypothesis is that there is a speedup when applying *O3* in favor of *O2* for 400.perlbench. To validate this claim, we use the paired t-test to see if there is any statistically significant difference in the means between the two distributions[6]. Indeed, at the 95% confidence interval we estimate the mean of *O2* divided by the mean of *O3* to be 1.007 ± 0.003, a slow down, because the fraction is greater than 1.0! Although tiny, the slow down is statistically significant, because the confidence interval falls above 1.0. We as researchers in systems *expect* that *O3* should provide a sizable increase in performance. However, when we factor out the effects due to shell environment state and linking order, we find that there is a very small decrease in performance for this benchmark.

While our approach explores more of the space of measurement context than a single run, it does not fully explore the space. Thus, while we consider our results to be more reliable than just comparing two runs in the same measurement context, our approach is also not definitive: a definitive approach would have to randomly sample contexts from *all* possible measurement contexts. We hope that as our approach gains use within the community, researchers will extend it to consider other sources of measurement bias and thus cover more of the space of measurement contexts.

The methods used in this paper are akin to how people operate in other sciences. For instance, imagine an epidemiologist hypothesizes that a certain food has the potential to cause a certain form of cancer. One would not expect her, for instance, to set up a test group made entirely of smokers—being a smoker increases the chance of developing a cancer. A proper experiment would realize smoking is a potential source of bias and mitigate its effects on the experiment's outcome by having an equal number of smokers in both the test and control groups. Moreover, this process of understanding those things that potentially add bias is ongoing. If, after the fact, a new source of bias is found that was not accounted for in a previously published result—the study may have to be redone.

Collectively, as a community, epidemiologists realize bias has the potential to impact the conclusions of their research. They have actively worked to develop methods to (i) isolate and determine factors that cause bias and (ii) systematically developed methods that allow a researcher to handle those factors. As we point out in this paper, in computer science experimental methods, bias is real, pervasive and sometimes able to overwhelm our conclusions. As a community we need to develop methods to overcome its implications—Indeed this paper is a first attempt at doing so.

To summarize, this section detailed an automated technique that allows a researcher to gain confidence that her conclusions are correct. If a source of bias is known, one can factor that bias out of any measurement by simply generating and measuring performance in many measurement contexts varying the bias. The results may still be biased due to unknown factors, but at least in this way a researcher has factored out what she *knows* to cause bias.

5.3 Summary

We have described and demonstrated two approaches for ensuring that poor quality data does not lead us to an invalid conclusion.

Our first approach uses causal analysis to confirm our hypothesis. While this is the definitive approach, it is difficult to use and is not automatic: it can require domain knowledge to manipulate causal relationships. In Section 5.1, we demon-

[6] See [6] for a nice discussion of this calculation.

strated how to use causal analysis to expose an an invalid conclusion from poor-quality data.

Our second approach attempts to avoid measurement bias by measuring a system in a large number of automatically generated variants of the measurement context. This approach helps greatly with measurement bias but it is limited by the variants that we generate automatically. In this paper we have identified two ways of generating variants: by changing linking order and by changing the size and settings of environment variables. While we have demonstrated that these are both important sources of measurement bias, these are not the only sources of measurement bias. We expect that as this technique gets more use, researchers will identify other sources of measurement bias and add them to this technique.

6. Related Work

Korn et al. [10] 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. [13] 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 PAPI (PAPI is the infrastructure we use for collecting hardware metrics). Moore [14] 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 [2], 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 [8] 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 [6, 9].

7. 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"; poor-quality data is data that can lead to invalid conclusions.

In this paper, we explore two common causes of poor-quality data. First, we illustrate that observer effect causes poor-quality data by dramatically perturbing the system that is being measured and may lead a performance analyst to draw invalid conclusions. For example, in Section 4.1 we show that something as benign as collecting software metrics with only a small increase in the number of instructions executed (less than 2%) can significantly perturb our system leading a performance analyst to believe, for example, that L1 instruction cache misses are a performance problem, when the cache misses are caused by the measurement infrastructure.

Second, we show that measurement bias can cause poor-quality data because the measurement context favors one system or variant of a system over another and may lead a performance analyst to draw invalid conclusions. For example, in Section 4.2 we show how even something as innocuous as increasing the size of a dummy UNIX environment variable can determine whether or not code compiled using "gcc -O3" is faster than code compiled using "gcc -O2".

We introduce techniques in Section 5 to detect poor-quality data: *causal analysis* detects when observer effect causes poor-quality data and *variant generation* detects when measurement bias causes poor-quality data. We demonstrate both techniques with case studies.

References

[1] *Proceedings of the International Symposium on Code Generation and Optimization (GCO 2007).* IEEE Computer Society.

[2] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*, pages 85–96, New York, NY, USA, 1997. ACM Press.

[3] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, 2006.

[4] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*, New York, NY, USA, October 2006. ACM Press.

[5] Rajagopalan Desikan, Doug Burger, and Stephen W. Keckler. Measuring experimental error in microprocessor simulation. In *SSR*, pages 266–277, New York, NY, USA, 2001. ACM Press.

[6] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA*, 2007.

[7] J.L. Hintze and R.D. Nelson. Violin plots: A box plot-density trace synergism. *The American Statistician*, 52(2):181–184, May 1998.

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

[9] Tomas Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *SPECTS*, pages 484–490, San Diego, CA, USA, 2005. SCS.

[10] W. Korn, P. J. Teller, and G. Castillo. Just how accurate are performance counters? In *IPCCC*, pages 303–310, 2001.

[11] Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. A loop correlation technique to improve performance auditing. *PACT*, 0:259–269, 2007.

[12] Han Lee, Daniel von Dincklage, Amer Diwan, and Eliot Moss. Understanding the behavior of compiler optimizations. *Software: Practice & Experience*, 2006. To appear.

[13] M. Maxwell, P. Teller, L. Salayandia, and S. Moore. Accuracy of performance monitoring hardware. In *LACSI*, October 2002.

[14] Shirley V. Moore. A comparison of counting and sampling modes of using performance monitoring hardware. In *ICCS*, pages 904–912,

London, UK, 2002. Springer-Verlag.

[15] George C. Necula, Scott McPeak, Shree Prakash Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In *CC*, pages 213–228, London, UK, 2002. Springer-Verlag.

[16] *Proceedings of the 16th International Conference on Parallel Architecture and Compilation (PACT 2007).* IEEE Computer Society.

[17] *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2007).*

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

[19] 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.

[20] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/.