Producing Wrong Data Without Doing Anything Obviously Wrong!

Todd Mytkowicz Amer Diwan
Department of Computer Science
University of Colorado
Boulder, CO, USA
{mytkowit,diwan}@colorado.edu

Matthias Hauswirth
Faculty of Informatics
University of Lugano
Lugano, CH
Matthias.Hauswirth@unisi.ch

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

Abstract

This paper presents a surprising result: changing a seemingly innocuous aspect of an experimental setup can cause a systems researcher to draw wrong conclusions from an experiment. What appears to be an innocuous aspect in the experimental setup may in fact introduce a significant bias in an evaluation. This phenomenon is called *measurement bias* in the natural and social sciences.

Our results demonstrate that measurement bias is significant and commonplace in computer system evaluation. By *significant* we mean that measurement bias can lead to a performance analysis that either over-states an effect or even yields an incorrect conclusion. By *commonplace* we mean that measurement bias occurs in all architectures that we tried (Pentium 4, Core 2, and m5 O3CPU), both compilers that we tried (gcc and Intel's C compiler), and most of the SPEC CPU2006 C programs. Thus, we cannot ignore measurement bias. Nevertheless, in a literature survey of 133 recent papers from ASPLOS, PACT, PLDI, and CGO, we determined that none of the papers with experimental results adequately consider measurement bias.

Inspired by similar problems and their solutions in other sciences, we describe and demonstrate two methods, one for detecting (causal analysis) and one for avoiding (setup randomization) measurement bias.

Categories and Subject Descriptors C. Computer Systems Organization [*C.4 Performance of Systems*]: Design studies

General Terms Experimentation, Measurement, Performance

Keywords Measurement; Bias; Performance

ASPLOS'09, March 7–11, 2009, Washington, DC, USA.

1. Introduction

Systems researchers often use experiments to drive their work: they use experiments to identify bottlenecks and then again to determine if their optimizations for addressing the bottlenecks are effective. If the experiment is biased then a researcher may draw an incorrect conclusion: she may end up wasting time on something that is not really a problem and may conclude that her optimization is beneficial even when it is not.

We show that experimental setups are often biased. For example, consider a researcher who wants to determine if optimization O is beneficial for system S. If she measures S and $S + O$ in an experimental setup that favors $S + O$, she may overstate the effect of O or even conclude that O is beneficial even when it is not. This phenomenon is called *measurement bias* in the natural and social sciences. This paper shows that measurement bias is commonplace and significant: it can easily lead to a performance analysis that yields incorrect conclusions.

To understand the impact of measurement bias, we investigate, as an example, whether or not *O3* optimizations are beneficial to program performance when the experimental setups differ. Specifically, we consider experimental setups that differ along two dimensions: (i) UNIX environment size (i.e., total number of bytes required to store the environment variables) because it affects the alignment of stack allocated data; and (ii) link order (the order of *.o* files that we give to the linker) because it affects code and data layout. There are numerous ways of affecting memory layout; we picked two to make the points in this paper but we have found similar phenomena with the others that we have tried.

We show that changing the experimental setup often leads to contradictory conclusions about the speedup of *O3*. By "speedup of *O3*" we mean run time with optimization level *O2* divided by run time with optimization level *O3*. To increase the generality of our results, we present data from two microprocessors, Pentium 4 and Core 2, and one simulator, m5 O3CPU [2]. To ensure that our results are not limited to gcc, we show that the same phenomena also appear when we use Intel's C compiler.

We show that there are no obvious ways of avoiding measurement bias because measurement bias is unpredictable. For example, the best link order on one microprocessor is often not the best link order on another microprocessor and increasing the UNIX environment size does not monotonically increase (or decrease) the benefit of the O3 optimizations. Worse, because hardware manufacturers do not reveal full details of their hardware it is unlikely that we can precisely determine the causes of measurement bias.

We show, using a literature survey of 133 recent papers from ASPLOS, PACT, PLDI, and CGO, that prior work does not carefully consider the effects of measurement bias. Specifically, to avoid measurement bias, most researchers use not a single workload, but a set of workloads (e.g., all programs from a SPEC benchmark suite instead of a single program) in the hope that the bias will statistically cancel out. For this to work, we need a 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 SPEC JVM98 benchmarks was not representative of typical Java applications [3]. We experimentally show that at least the SPEC CPU2006 (CINT and CFP, C programs only) benchmark suite is not diverse enough to eliminate the effects of measurement bias.

Finally, this paper discusses and demonstrates one technique for avoiding measurement bias and one technique for detecting measurement bias. Because natural and social sciences routinely deal with measurement bias, we derived two techniques directly from these sciences. The first technique, *experimental setup randomization* (or *setup randomization* for short), runs each experiment in many different experimental setups; these experiments result in a distribution of observations which we summarize using statistical methods to eliminate or reduce measurement bias. The second technique, *causal analysis* [16], establishes confidence that the outcome of the performance analysis is valid even in the presence of measurement bias.

The remainder of the paper is structured as follows. Section 2 explores the origin of measurement bias. Section 3 presents our experimental methodology. Sections 4 and 5 show that measurement bias is significant, commonplace, and unpredictable. Section 6 shows that prior work inadequately addresses measurement bias. Section 7 presents techniques for dealing with bias. Section 8 discusses what hardware and software communities can do to help with measurement bias. Section 9 compares this paper to related work, and Section 10 concludes.

2. Origin of Measurement Bias

Program performance is sensitive to the experimental setup in which we measure the performance. For example, we measured the execution time of the code in **Figure 1**(a) while changing an innocuous part of the experimental setup; the UNIX environment size. Figure 1(b) shows the outcome of this experiment. A point (x, y) says that when the UNIX environment size is x bytes, the execution time is y cycles on a Core 2 workstation.[1] Each point represents the mean of five runs and the whiskers represent the 95% confidence interval of the mean. We see that something external and orthogonal to the program, i.e., changing the size (in bytes) of an unused environment variable, can dramatically (frequently by about 33% and once by almost 300%) change the performance of our program. This phenomenon occurs because the UNIX environment is loaded into memory before the call stack. Thus, changing the UNIX environment size changes the location of the call stack which in turn affects the alignment of local variables in various hardware structures.

This simple example demonstrates that computer systems are sensitive: an insignificant and seemingly irrelevant change can dramatically affect the performance of the system. As a consequence of this sensitivity, we will find that different experimental setups will produce different outcomes and thus cause measurement bias.

3. Experimental Methodology

In a comparison between two systems, S_1 and S_2, measurement bias arises whenever the experimental setup favors S_1 over S_2 or vice versa. Thus, measurement bias can make it appear that one system (e.g., S_1) is superior to another system (e.g., S_2) even when it is not.

Measurement bias is well known to medical and other sciences. For example, Ioannidis [9] reports that in a survey of 49 highly-cited medical articles, later work contradicted 16% of the articles and found another 16% had made overly strong claims. The studies that contradicted the original studies used more subjects and random trials and thus probably suffered less from measurement bias.

This paper considers two sources of measurement bias: (i) the UNIX environment size which affects the start address of the stack and thus stack allocated data alignment; and (ii) link order which affects code and data layout. We picked these sources because it is well known that program performance is sensitive to memory layout and thus anything that affects memory layout is also likely to exhibit measurement bias.

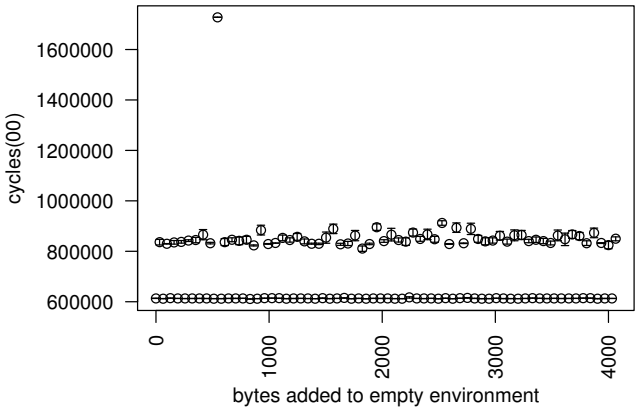
There are numerous other sources of measurement bias. For example the room temperature affects the CPU clock speed and thus whether the CPU is more efficient in executing memory-intensive codes or computationally-intensive codes [6]. As another example, the selection of benchmarks also introduces measurement bias; a benchmark suite whose codes have tiny working sets will benefit from different optimizations than codes that have large working sets. It is

[1] We disabled all optimizations for this kernel to keep it simple; this is why the compiler does not eliminate the loop as dead code. A slightly more complex version of this example exhibits the same phenomenon with optimizations enabled.

```
static int i = 0, j = 0, k = 0;

int main() {
  int g = 0, inc = 1;
  for (; g<65536; g++) {
    i += inc;
    j += inc;
    k += inc;
  }
  return 0;
}
```

(a) C code for micro-kernel



(b) Effect of environment variable size on performance.

Figure 1. A micro-kernel that shows extreme sensitivity

Suite	Benchmark	Cycles	Seconds
CINT	gcc	3,142,640,332	1.3
	libquantum	7,690,324,238	3.2
	perlbench	12,752,930,486	5.3
	bzip	13,840,302,589	5.8
	h264ref	46,785,473,888	19.5
	mcf	59,250,579,566	25.0
	gobmk	180,491,870,344	75.2
	hmmer	246,974,548,791	102.9
	sjeng	419,892,937,630	175.0
CFP	sphinx3	35,238,100,682	14.7
	milc	57,414,647,507	23.9
	lbm	232,213,767,693	96.8

Table 1. Benchmarks used in our experiments.

not the goal of our paper to expose all sources of measurement bias; instead, the goal is to show that measurement bias exists and one needs to use techniques such as the ones described in Section 7 to deal with it.

In the sections that follow we explore the speedup of gcc's *O3* optimizations over *O2*. We should stress that it is not the point of the paper to evaluate whether one optimization level is better than another, rather we use this experiment to show how easily measurement bias can affect our conclusions. The remainder of this section describes our experimental setup.

3.1 Benchmarks

Table 1 shows the benchmarks from the SPEC CPU2006 V1.0 [17] benchmark suite we use to explore measurement bias. We use all the C benchmarks from the CINT (integer) and CFP (floating point) components of SPEC CPU2006. We did not consider the non-C programs because the optimization levels would not be comparable between the compilers for different programming languages. All of these benchmarks are single-threaded applications. To generate the data for the figures in this paper, we ran each benchmark 5,940 times. To ensure that we could collect this data in a timely manner, we used the *train* input for all experiments. For each benchmark, the "Cycles" column in the table gives the mean runtime (in cycles) of 15 runs using an empty UNIX environment and default link order. Even with the train input, it took us 12 days to generate the needed data for our longest running benchmark, sjeng.

3.2 Measurement infrastructure

We conducted our experiments on two machines: a Pentium 4 and a Core 2 workstation (**Table 2**). On both machines we ran Linux and used PAPI [5] to extract hardware performance monitor information. We added all our instrumentation that accesses PAPI in an inter-loper (a shared library that overloads `__libc_start_main` via LD_PRELOAD). The inter-loper sets up the data collection before `main` executes and reads out the collected data after `main` exits, ensuring that measurement overhead is minimized. Adding instrumentation in this way does not alter the static memory layout of a program as it does not require source instrumentation or recompilation of a benchmark.

Unless we state otherwise, all data in this paper is for the Core 2 workstation. We picked the Core 2 because it is a widely deployed state-of-the-art processor. The Core 2 workstation we used was a quad core; however, our single threaded benchmarks only used a single core at any given time. We report selected data for the Pentium 4 workstation and the m5 simulator using the O3CPU model [2] to demonstrate the generality of our results.

For each configuration, or experimental setup, we run the benchmark multiple times, back-to-back. If there is OS activity that persists for all the runs of a configuration, it may bias our results. However, we are certain this is not the case for two reasons. First, in several cases we did run additional runs of a configuration to explore some phenomenon in more detail and got reproducible results; if OS activity was causing bias this would not have been the case. Second, even for our shortest running benchmark the OS activity would have to persist for over 6.5 seconds to bias our results and much longer (minutes) for the longer running benchmarks; this is unlikely.

Parameter	Core 2	Pentium 4	m5 O3CPU
Operating System	Linux 2.6.25	Linux 2.4.21	NA
Tool Chain	gcc 4.1.3, icc 10.1	gcc 4.2.1	gcc 4.1.0
Measurement	papi-3.5.1 / perfmon-2.8	papi-3.0.8 / perfctr-5.2.16	NA
Micro-architecture	Core	NetBurst	Alpha
Clock Frequency	2.4 GHz	2.4 GHz	1GHz
memory	8G	2G	512M
L1	32K Ins., 32K Data	12K Ins. 8K Data	32K Ins. 64K Data
L2	128K Unified	512K Unified	2M Unified
L3	4096K	NA	NA
TLB entries	512	64	48 Ins. 64 Data

Table 2. Description of the machines used in our study to show the effects of measurement bias.

3.3 Following best practices

With all aspects of our measurements we attempted to be as careful as possible. In other words, the measurement bias that we demonstrate later in the paper is present despite our following best practices.

- 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 Core 2) 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.

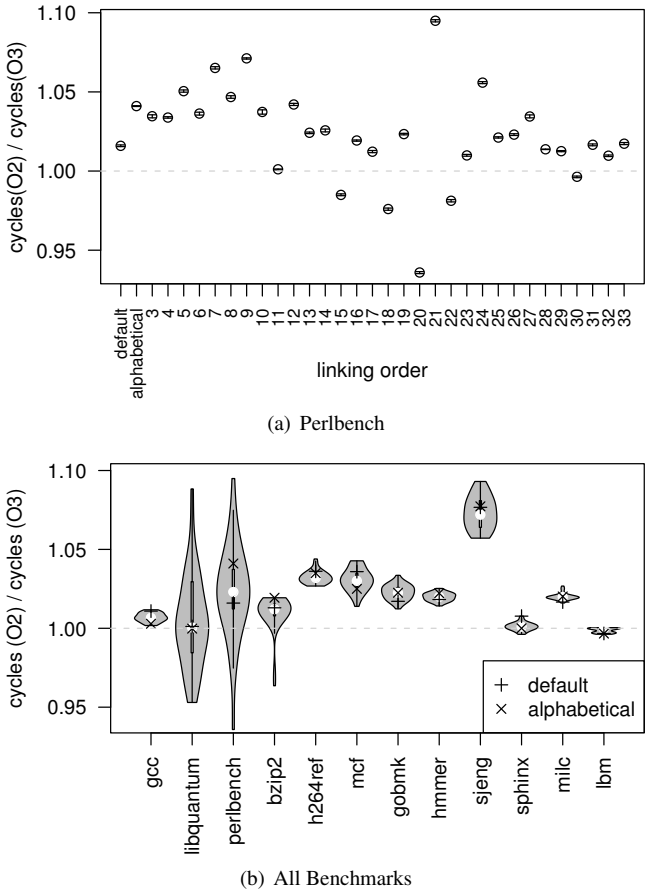
4. Measurement Bias is Significant and Commonplace

This section shows that measurement bias is significant and commonplace. By significant we mean that measurement bias is large enough to lead to incorrect conclusions. By commonplace we mean that it is not an isolated phenomenon but instead occurs for all benchmarks and architectures that we tried.

We quantify measurement bias with respect to the following question: how effective are the *O3* optimizations in gcc? By "*O3* optimizations" we mean optimizations that *O3* introduces (i.e., it does not include optimizations that carry over from *O2*).

4.1 Measurement bias due to link order

We first show the measurement bias due to link order for all benchmarks and then discuss one potential cause for it on one benchmark.



(a) Perlbench

(b) All Benchmarks

Figure 2. The effect of link order on Core 2.

4.1.1 The extent of measurement bias

Figure 2 (a) explores the effect of link order on the speedup of *O3* for perlbench. To obtain this data, we compiled perlbench 33 times; the first time we used the default link order (as specified by the make file), the second time we used an alphabetical link order (i.e., the .o files appeared in alphabetical order), and the remaining times we used a randomly generated link order. A point (x, y) in Figure 2 (a) says that for the x^{th} link order we tried, the speedup of *O3* was y. For each point, we conducted five runs each with *O2* and *O3*; the whiskers give the 95% confidence intervals of the mean.

There are three important points to take away from this graph. First, depending on link order, the magnitude of our conclusion (height of y-axis) can significantly fluctuate (0.92 vs 1.10). Second, depending on link order, *O3* either gives a speedup over *O2* (i.e., y value is greater than 1.0) or a slow down over *O2* (i.e., y value is less than 1.0). Third, some randomly picked link orders outperform both the default and alphabetical link orders. Because we repeated the experiment for each data point multiple times, these points are not anomalies but true, reproducible behavior.

Figure 2 (b) uses a violin plot to summarize similar data for all benchmarks. Each violin summarizes data for all the link orders for one benchmark; e.g., the perlbench violin summarizes Figure 2 (a). The white dot in each violin gives the median and the thick line through the white dot gives the inter-quartile range. The width of a violin at y-value y is proportional to the number of times we observed y. The "+" and "$\times$" points in each violin give the data for the default and alphabetical link orders respectively.

From Figure 2 (b) we see that the violins for five benchmarks (libquantum, perlbench, bzip2, sphinx and lbm) straddle 1.0; thus, for these benchmarks, we may arrive at conflicting conclusions about the benefit of the *O3* optimizations depending on the link order that we use. On the Pentium 4 the results are more dramatic: *all* of the violins for the non-FP benchmarks straddle 1.0[2].

In Figure 2 (b) the differences between the maximum and minimum points of a violin (the height of the violin) are particularly instructive because they give an indication of the range of bias one can end up with. On the Core 2 for benchmark perlbench, the difference between the maximum and minimum points of the violin is 0.15. In other words we may think we have a 7% slowdown when in fact we have a 8% speedup! To generalize these results, the median difference between the minimum and maximum points is 0.02 while on the Pentium 4 the median difference is 0.08. Thus, the measurement bias due to link order is significant: we can arrive at significantly different (on average 2% for Core 2 and 8% for Pentium 4, depending on our experimental setup.

We repeated a selection of the above experiments on the m5 simulator using the O3CPU model [2]. We used smaller inputs for the m5 experiments because the train inputs took too long to simulate. We found that changing link order also caused measurement bias on the simulator; for example for bzip2 the speedup of *O3* ranged from 0.8 to 1.1 as a result of different link orders. Thus, measurement bias due to link order is commonplace: we found it on all three machines (one simulated) and all the benchmarks that we tried.

4.1.2 The potential causes of measurement bias

What causes measurement bias due to link order on the Core 2? Changing a program's link order can affect performance in a number of ways. For example, link order affects the alignment of code, causing conflicts within various hardware buffers (e.g. caches) and hardware heuristics (e.g. branch prediction). The link order may affect different programs differently; in one program it may affect the alignment of code in the instruction queue and in another program it may affect conflict misses in the instruction cache.

To investigate why we see these effects we used the m5 simulator (O3CPU model) as we have full knowledge of its internals (via its source code). We investigated the bzip2 benchmark and found that whether or not a particular hot loop fit in a single cache line determined the performance of the program. If the loop fit entirely in a cache line, the m5 O3CPU simulator latches it in the instruction queue and thus avoids any i-cache accesses for the duration of the loop; thus the loop executes faster than if it did not fit in a single cache line.

To understand if the fluctuations on the real hardware also had the same explanation, we tried to identify features on real hardware that would also avoid i-cache accesses for such loops. We found that at least the Core2 has such a feature: the loop-stream detector (LSD).

While we know that alignment of code can affect whether or not a hot loop fits in the LSD and that the alignment of code changes when we change the link order, three reasons prevent us from confirming this explanation. First, Intel does not release the full description of the LSD so we cannot look at a loop and know for sure whether or not it would benefit from the LSD. Second, Intel does not provide any hardware performance monitors that directly measure the behavior of the LSD. Third, since the LSD is always on (e.g., we cannot disable it as we can disable some hardware features such as prefetching) we cannot simply turn it off and see if the performance fluctuations disappear.

More generally, we find that inadequate information from hardware manufacturers and from the hardware severely cripples our ability to (i) understand the performance of a system and to (ii) fully exploit the capabilities of the hardware.

4.2 Measurement bias due to UNIX environment size

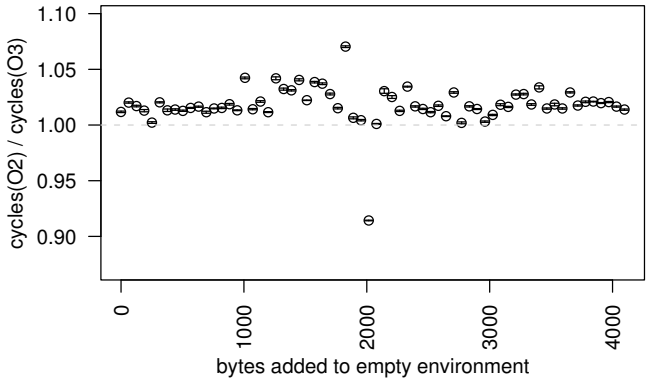
We first show the measurement bias due to environment variables and then discuss two potential causes for it.

4.2.1 The extent of the measurement bias

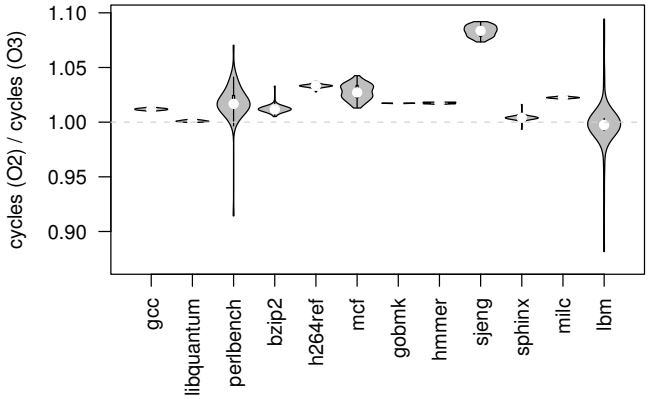
Figure 3 (a) shows the effect of UNIX environment size for perlbench on the speedup of *O3*. The leftmost point is for a shell environment of 0 bytes (the null environment); all subsequent points add 63 bytes to the environment. To increase the UNIX environment size, we simply extend the string value of a dummy environment variable that is not used by the program.

A point (x, y) says that when the UNIX environment size is x bytes, the speedup of *O3* is y. We computed each point using five runs each with *O2* and *O3*; the error bars give

[2] We did not collect the data for the three SPEC CPU2006 CFP benchmarks on the Pentium 4.



(a) Perlbench



(b) All Benchmarks

Figure 3. The effect of UNIX environment size on the speedup of *O3* on Core 2.

the 95% confidence intervals around the mean. The tight confidence intervals mean that these points are not anomalies but true reproducible behavior.

The most important point to take away from this graph is that depending on the shell environment size we may conclude that (i) the *O3* optimizations are beneficial (i.e., the y value is greater than 1.0); (ii) the *O3* optimizations degrade performance; and (iii) increasing the UNIX environment size does not predict the *O3* speedup because as the size increases speedup increases and decreases, ranging from 0.91 to 1.07.

Figure 3 (b) 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 a particular UNIX environment size).

In Figure 3 (b), we see that four of the violins (libquantum, perlbench, sphinx and lbm) straddle 1.0: this means that depending on the experimental setup, one can end up with contradictory conclusions about the speedup of *O3*. On the Pentium 4, our results are more dramatic: the violins of *six* of the 9 CINT benchmarks straddle 1.0.

The perlbench violin summarizes Figure 3 (a). The difference between the maximum and minimum points of a violin are particularly instructive because they give an indication of the range of bias one can end up with. The most extreme is lbm which ranges from 0.88 (i.e., a significant slowdown due to *O3* optimizations) to 1.09 (i.e., a healthy *O3* speedup). The median difference between the extreme points on the Core 2 is 0.01 and on the Pentium 4 is 0.04. Thus, while smaller than measurement bias due to link order, the measurement bias due to UNIX environment size is still large enough (on average 1% for Core 2 and 4% for Pentium 4) to obfuscate experimental results.

To summarize, measurement bias due to UNIX environment size can severely impact the results of our experiment—both in the magnitude of any effect we are measuring and even forcing a researcher to draw an incorrect conclusion. Using lbm as an example we may think we have a 12% slowdown when in fact we have a 9% speedup!

4.2.2 The potential causes of the measurement bias

What causes the measurement bias due to environment variables on the Core 2? So far we have uncovered two high-level reasons.

The first reason is that the UNIX environment size affects the starting address of the C stack. Thus, by changing the UNIX environment size, we are effectively changing the address and thus the alignment of stack variables in various hardware buffers; also many algorithms in hardware (e.g., to detect conflicts between loads and stores) depend on alignments of code or data. We verified our explanation by always starting the stack at the same location while changing the UNIX environment size; we got the same *O3* speedup (for all benchmarks except perlbench) with different UNIX environment sizes, thus confirming that it was the stack starting location that affected *O3* speedup.

The second reason (which applies only to perlbench) is that when perlbench starts up, it copies contents of the UNIX environment to the heap. Thus, using different UNIX environment sizes effectively changes the alignment of heap-allocated structures in various hardware buffers in addition to the alignment of stack allocated variables. We confirmed this explanation by always fixing the start address of the heap so that all of our different UNIX environments would fit below it. With these experimental setups, we found that different UNIX environment sizes had a much smaller impact on the speedup of *O3*. The first reason described above (i.e., UNIX environment size affects stack start address) causes the residual bias.

While the above two reasons provide a high-level causal analysis, we would like to understand the underlying causes in more detail. In particular we would like to know *which hardware structure* interacted poorly with *which stack variables*. For this study we intervened on the code of perlbench and fixed the heap start address so as to focus entirely on the effects due to shifting the stack address. We picked the two UNIX environment sizes that lead to the fastest and the slowest execution time. For both of these UNIX environ-

ments we ran perlbench multiple times in order to capture all the Core 2 performance events provided by perfmon. Out of these 340 events, 42 events differed by more than 25%. One event stood out with a 10-fold increase in its event count: LOAD_BLOCK:OVERLAP_STORE, the number of loads blocked due to various reasons, among them loads blocked by preceding stores.

At a high level, we found that the alignment of stack variables was probably causing measurement bias. At a low level, we found that the LOAD_BLOCK:OVERLAP_STORE was probably causing measurement bias. What is the connection? The hardware uses conservative heuristics based on alignment and other factors to determine load-store overlaps. By changing the alignment of stack variables, we have probably affected the outcome of the hardware heuristics, and thus the number of load-store overlaps.

To increase our confidence in the hypothesis that the load-store overlap is responsible for the measurement bias, we would need further support from the hardware. In particular, if we could map events to program locations, we could determine whether these load blocks happen due to stack accesses, and we would be able to point out which stack locations are responsible for these conflicts. The Core 2 already provides such support through precise event based sampling (PEBS). Unfortunately, PEBS supports only a very limited set of events, and LOAD_BLOCK:OVERLAP_STORE is not part of that set.

4.3 Did gcc cause measurement bias?

So far our experiments all used the gcc compiler. If gcc does not take the alignment preferences of the hardware into consideration, perhaps because the hardware manufacturers do not reveal these preferences, then code compiled with gcc may be more vulnerable to measurement bias. Thus, we repeated our experiments with Intel's ICC compiler; we expected that the Intel compiler would exhibit little measurement bias. We were wrong.

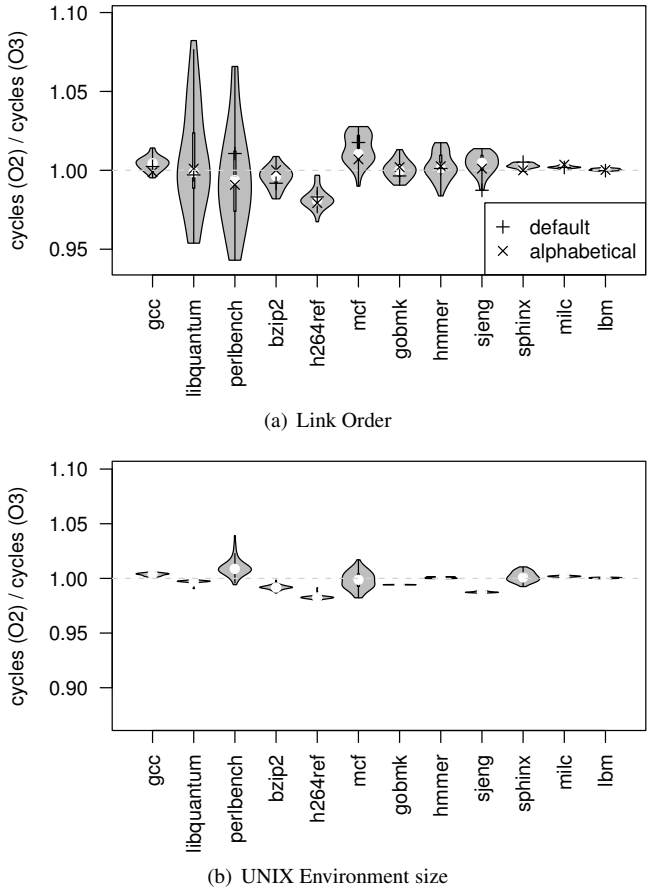
Figure 4 (a) presents data similar to Figure 2 (b) and **Figure 4 (b)** presents data similar to Figure 3 (b) except that it uses Intel's C compiler instead of gcc. We see that we get measurement bias also with Intel's C compiler. For our experiments with link order, the violins for 10 (6 for gcc) of the benchmarks straddle 1.0 and the median height of the violins is 0.03 (0.02 for gcc). For our experiments with UNIX environment size, 6 (4 for gcc) of the violins straddle 1.0 and the median height of the violins is 0.006 (it was 0.01 with gcc). Thus, code compiled with Intel's C compiler also exhibits measurement bias.

4.4 Summary

We have shown that measurement bias is significant and commonplace.

Measurement bias is significant because it 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 experimental setup she would have concluded the exact opposite.

Measurement bias is commonplace because we have observed it for all of our benchmark programs, on three microprocessors (one of them simulated), and using both the Intel and the GNU C compilers.



(a) Link Order

(b) UNIX Environment size

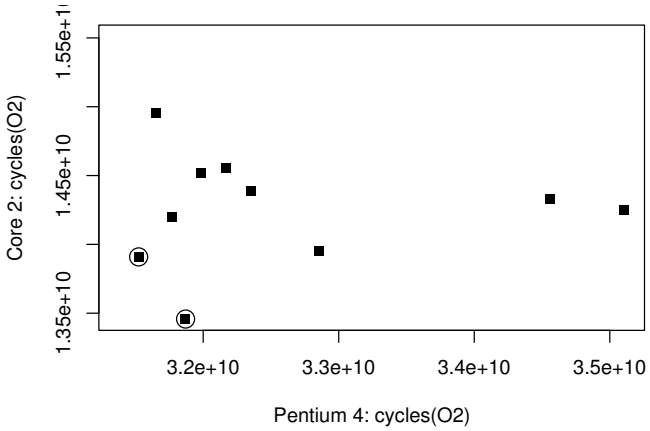
Figure 4. Bias in Intel's C compiler on Core 2.

5. Measurement Bias is Unpredictable

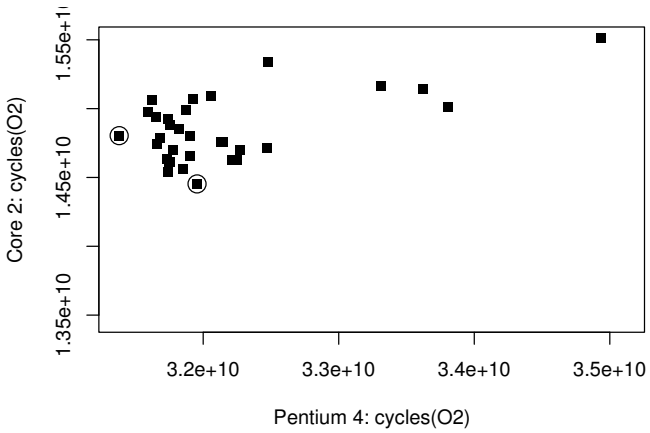
If measurement bias is predictable then it should be easy to avoid. Unfortunately, we found that measurement bias is not easily predictable.

In Figure 3 (a) we saw for perlbench that increasing the UNIX environment size can make *O3* appear better or worse. Thus, increasing the UNIX environment size does not always translate to consistently more (or less) bias.

Moreover, **Figure 5** shows for perlbench that measurement bias on one machine does not predict measurement bias on another machine. **Figure 5 (a)** presents data for link order. A point (x,y) says that there is a link order that gives an execution time x on the Pentium 4 and an execution time y on the Core 2. The two circled points represent the best link order for each of the machines: The leftmost circled point represents the best link order for Pentium 4 and the



(a) Link Order



(b) UNIX Environment size

Figure 5. Measurement bias in perlbench.

bottom-most circled point represents the best link order for the Core 2. Because the points are not the same, the best link order for one machine is not the best link order for the other.

This insight is of particular interest to the software vendors who distribute their software already linked. If they tune link order for a particular machine, there is no guarantee that the link order will provide the best performance on a different machine.

Figure 5 (b) presents data for UNIX environment size. A point (x,y) says that there is a UNIX environment size that gives an execution time x on the Pentium 4 and an execution time y on the Core 2. The two circled points represent the best UNIX environment size for each of the machines: the leftmost circled point represents the best for Pentium 4 and the bottom-most circled point represents the best for the Core 2. Because the points are not the same, the best UNIX environment size for one machine is not the best for the other.

In summary, the UNIX environment size does not predict performance. Furthermore, the best link order or best UNIX environment size on one machine is not necessarily the best on another machine.

6. Literature Review

Researchers in computer systems either do not know about measurement bias or do not realize how severe it can be. For example, we found that none of the papers in APLOS 2008, PACT 2007, PLDI 2007, and CGO 2007 address measurement bias satisfactorily.

We picked these conferences because they are all highly selective outlets for experimental computer science work. Of the 133 papers published in the surveyed conference proceedings, 88 had at least one section dedicated to experimental methodology and evaluation. The remainder of this review focuses on these 88 papers. When something was not clear in a paper, we always gave the benefit of the doubt to a paper's methodology.

6.1 Papers that use simulations

Many researchers use simulations because simulators enable them to try out hypothetical architectures. 36 of the 88 papers we reviewed used simulations. As we have shown in Section 4.1, even simulations suffer from measurement bias.

6.2 Papers that report speedups

If the ideas in a paper result in huge (e.g., many-fold) improvements, then one may argue that the improvements are a result of the ideas and not artifacts of measurement bias. However, we found that the median speedup reported by these papers was 10%; in Section 4 shows a measurement bias large enough to easily obsfucate a 10% speedup.

6.3 Papers that acknowledge measurement bias

In this paper, we focus on two specific instances of measurement bias (UNIX environment size and link order) and demonstrate that they can cause invalid conclusions. Although none of the papers we reviewed said anything about measurement bias due to UNIX environment size or link order, most (83) papers used more than one benchmark (with a mean number of benchmarks being 10.6 ± 1.8) or input sets for their evaluation. If we use a sufficiently diverse set of workloads along with careful statistical methods, most measurement bias should get factored out. However, as we show in Section 7.1.1, this is a partial solution; significant measurement bias may remain even with a large benchmark suite. Indeed even with our 12 benchmarks we still see large measurement bias.

7. Solutions: Detecting and Avoiding Measurement Bias

Measurement bias is not a new phenomenon and it is not limited to computer science. To the contrary, other sciences have routinely dealt with it. For example, consider a study that predicts the outcome of a nationwide election by polling only a small town. Such a study would lack all credibility because it is biased: it represents only the opinions of a set of (probably) like-minded people. This is analogous to evalu-

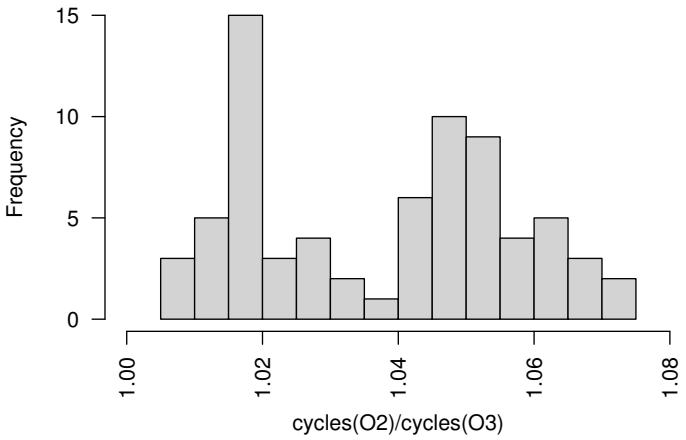


Figure 6. Distribution of speedup due to *O3* as we change the experimental setup.

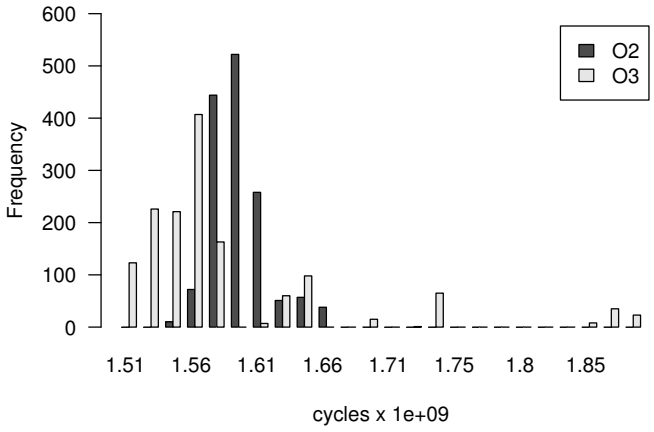


Figure 7. Using setup randomization to determine the speedup of *O3* for perlbench. At a 95% confidence interval, we estimate the speedup to be 1.007 ± 0.003.

ating an optimization in only one or a small number of experimental setups. Given that our problem is an instance of something that other sciences already deal with, our solutions are also direct applications of solutions in other sciences.

7.1 Evaluate innovations in many experimental setups

The most obvious solution to the polling-a-small-town problem is to poll a diverse cross section of the population. In computer systems we can do this by using a diverse set of benchmarks or by using a large set of experimental setups or both.

7.1.1 Using a large benchmark suite

If we use a sufficiently 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, SPEC JVM98, was less memory intensive than Java workloads in recent years [3]. As a consequence, the SPEC JVM98 benchmarks may be biased against virtual machines with sophisticated memory managers.

Figure 6 evaluates whether or not our benchmark suite (see Table 1) is diverse enough to factor out measurement bias due to memory layout. To generate this figure, we measured the speedup of *O3* for all benchmarks in 66 different experimental setups; these setups differ in their memory layouts. For each setup we generated one number: this number is the average speedup of the entire suite for that experimental setup. Figure 6 plots the distribution of these average speedups; specifically the height of a bar at x-value x gives the number of experimental setups when we observed the average speedup x.

If our benchmark suite was diverse enough to factor out measurement bias, we would see a tight distribution; in other words, varying the experimental setup would have had little impact on the average speedup across the benchmark suite. Instead, we see that there is a 7% variation in speedup between different experimental setups. Thus, our benchmark suite is not diverse enough to factor out measurement bias due to memory layout.

While our results are disappointing, they do not preclude the possibility that a well designed benchmark suite may factor out measurement bias due to memory layout. While we may be tempted to create such a suite by combining existing benchmark suites, we should point out that it is not the size of the benchmark suite that is important; it is the diversity of the suite that determines whether or not the suite can factor our measurement bias.

7.1.2 Experimental setup randomization

In this approach, we generate a large number of experimental setups by varying parameters that we know to cause measurement bias. Thus we measure the systems being compared, S_1 and S_2, in each of these experimental setups. This process results in two distributions: one for S_1 and one for S_2. Finally, we use statistical methods, such as the t-test, to compare the distributions to determine if S_1 is better than S_2.

Figure 7 shows the distributions we get when we vary link order and UNIX environment size for the perlbench benchmark. For this figure we used 484 measurement setups (using a cross product of 22 linking orders and 22 environment variable sizes). We conducted 3 runs for each combination of experimental setup and benchmark to mitigate the effects of inter-run variation. We then used the t-test to see if there is any statistically significant difference in the means

between the two distributions[3]. Using this test we found that at the 95% confidence interval the mean speedup ratio was 1.007 ± 0.003; in other words, *O3* optimizations speed up the program.

While this approach is certainly better than using just a single experimental setup, it is not perfect: if we do not vary the experimental setup adequately then we may still end up with measurement bias for a different bias. This is one of the reasons why in other sciences it is not uncommon to find contradictions. For example Ioannidis [9] reports that as many has 32% of the most high-profile studies (more than 1,000 citations) in medicine were later found to be either incorrect or exaggerated; the later studies used larger sample sizes and thus, because they used randomized trials, presumably less bias.

7.2 Using causal analysis

Causal analysis is a general technique for determining if we have reached an incorrect conclusion from our data [16]. The conclusion may be incorrect because our data is tainted or because we arrived at the conclusions using faulty reasoning.

At an abstract level, let's suppose we arrive at the following conclusion: X caused Y. Now, it may be the case that there are many other possible causes of Y (e.g., Z); so we wish to check whether or not our conclusion is valid. To achieve this, causal analysis takes the following steps:

1. *Intervene*: We devise an intervention that affects X while having a minimal effect on everything else.
2. *Measure*: We change our system with the intervention and measure the changed system.
3. *Confirm*: If Y changed as we would expect if X had caused Y then we have reason to believe that our conclusion is valid.

Earlier in the paper we have already seen examples of such reasoning. For example, we had arrived at the conclusion: Changing UNIX environment size *causes* a change in start address of the stack which *causes* a change in *O3* speedup.

We were easily able to verify the first "causes" by noting that when we changed UNIX environment size by b bytes, the address of stack variables also shifted by b bytes (modulo appropriate alignment). For the second "causes" we had to use causal analysis as follows:

1. *Intervene*: We fixed the starting address of the stack so that regardless of the UNIX environment size (up to 4096 bytes) the stack always started at the same address.
2. *Measure*: We varied the UNIX environment size (up to 4096 bytes) and calculated the *O3* speedups for each environment size.
3. *Confirm*: We confirmed that the UNIX environment size did not affect *O3* speedup, thus giving credibility to our conclusion.

Using the above analysis, we were able to confirm our conclusions for all benchmarks except perlbench. For perlbench our causal analysis rejected our conclusion and we had to come up with a different conclusion which we then confirmed. There are typically many ways of conducting a causal analysis. For example, we could have picked a different intervention: change the starting address of the stack while always using the same environment variables. Also, while causal analysis gives us confidence that our conclusions are correct, it does not guarantee them; this is a fact of life that all experimental sciences have to contend with.

In contrast to setup randomization, causal analysis is not an attempt to get "untainted" data; instead it is a way to gain confidence that the conclusions that we have drawn from our data are valid even in the presence of measurement bias.

In the context of this paper, the conclusions that we have been exploring are of the form: "*O3* optimizations improve performance". To apply causal analysis we may need to modify the optimizations so that we can determine if the performance improvement is due to the optimization; and not due to a lucky interaction between the optimization and the experimental setup.

7.3 Summary

We have described and demonstrated two approaches: one for avoiding and one for detecting measurement bias. Our first approach is to collect data in not one but many (varied) experimental setups and then use statistical techniques to factor out measurement bias from the data. This approach actually tries to avoid measurement bias. Our second approach is to use causal analysis to check the validity of conclusions we draw from the data. This approach detects when measurement bias has led us to an incorrect conclusion.

Neither of these techniques are perfect. For example, even if we use a large number of experimental setups we may still not adequately cover the space of possible experimental setups. This problem, however, is not surprising: natural and social sciences also routinely deal with measurement bias using the same techniques we propose and they too find that occasionally even with the best methodology they end up with incorrect conclusions.

8. A Call to Action

We have shown that measurement bias poses a severe problem for computer systems research. While we have also presented approaches for detecting and avoiding measurement bias, these techniques are not easy to apply; this section discusses what the software and hardware communities can do to make these techniques more easily and widely applicable.

[3] See Georges *et. al* [7] or most statistics texts for a description of this calculation.

8.1 Diverse Evaluation Workloads

If we conduct our measurements over diverse workloads and use statistical methods to draw conclusions from these measurements we will reduce or perhaps even avoid measurement bias in our data. Section 7.1.1 shows that at least the C programs in the SPEC CPU2006 benchmark suite are not diverse enough to avoid measurement bias. Efforts, such as the Dacapo benchmarks [3], go to some length to ensure diversity within the suite; we need more efforts like this for different problem domains and programming languages.

8.2 Identify more ways of randomizing experimental setup

It is well known that memory layout impacts performance; this is why we varied the memory layout to generate different experimental setups. However, there are other features of the experimental setup that also cause measurement bias; indeed Section 9 points out that each problem domain introduces its own sources of measurement bias[18, 4, 1]. The natural and social sciences, based on long experience, have identified many sources of measurement bias in their domains; e.g., gender and education are both sources of measurement bias when we are trying to predict the outcome of a presidential election. We need to go through the same process and use that knowledge to build tools that automatically generate randomized experimental setups; this way a systems researcher can start from a good baseline when conducting her experiments.

8.3 More information from the hardware manufacturers

If we do not know the internal details of a microprocessor, it can be nearly impossible to (i) fully understand the performance of even a micro-kernel running on the microprocessor; and (ii) fully exploit the microprocessor to obtain peak performance. Sun Microsystems has already taken the lead by releasing all details of one of their microprocessors (the OpenSparc project); we hope other manufacturers will follow.

8.4 More cooperation from the hardware

Especially for causal analysis it is helpful if we can (i) collect data from major components of the microprocessor (e.g., caches); (ii) enable and disable optional hardware features; and (iii) map hardware events back to software.

Regarding (i), all modern microprocessors support hardware performance monitors, which allow us to collect data from some components of the hardware. Unfortunately, these metrics are often inadequate: for example, the Core 2 literature advertises the LSD as a significant innovation but fails to include any support for directly collecting data on the performance of this feature. We hope that in the future hardware will include metrics for at least all of the major components of the hardware.

Regarding (ii), some microprocessors allow us to enable or disable certain features. For example, the Core 2 allows us to enable and disable some forms of prefetching. However, there are many features that we cannot control in this way; for example there is no way to disable the LSD. We hope that in the future hardware will allow us to disable many more features.

Regarding (iii), precise event based sampling (PEBS) is invaluable: it enables us to map certain events to specific instructions. However, PEBS support is still overly limited; for example, Table 18.16 of Intel's Software Developer's manual [8] lists only nine events that PEBS supports. We hope that in the future hardware will allow us to map all events to specific instructions.

9. Related Work

Prior work has looked at various sources of faulty performance analysis.

Korn et al. [12] 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. Maxwell et al. [14] extends this work to three new platforms, POWER3, IA-64, and Pentium. Moore [15] discusses accuracy issues when using PAPI for counting events (PAPI is the infrastructure we use for collecting hardware metrics). In contrast to these papers which focus on performance counter accuracy using micro-kernels, our paper studies different sources of faulty performance analysis using real programs.

Most fields of experimental science use inferential statistics [10] 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 [7, 11].

Tsafrir et al. [18] describe "input shaking", a technique for determining the sensitivity of their simulations of parallel systems. Blackburn et al. [4] demonstrate that it is important to evaluate garbage collectors using many and not just a single heap size. Alameldeen and Wood[1] introduce variability in the latency of cache misses in order to alter thread-scheduling bias introduced by simulators that execute multi-threaded workloads. These techniques are examples of additional methods we could use (in addition to link order and environment variables) to generate different experimental setups. Each of these papers present sources of measurement bias that are specific to particular domains (simulations of parallel systems and garbage collected systems respectively); indeed each problem domain may have its own sources of measurement bias. This underlines our point that

measurement bias is not something that we can just eliminate: we may be able to eliminate some kinds of bias (e.g., by aligning loops in certain ways) but we will never be able to eliminate all kinds of bias. Thus, the techniques in Section 7 are not stop-gap measures to be used until we eliminate this problem; indeed the problem of measurement bias is here to stay and techniques such as those presented in this paper will be useful in dealing with it.

10. Conclusions

Would you believe us if we told you: "we can predict a national election by polling only a small town?" You should not: a small town probably contains a biased sample of the national population and thus one cannot draw nationwide conclusions from it.

Would you believe us if we told you: "we can predict the benefit of our optimization, O, by evaluating it in one or a few experimental setups using a handful of benchmarks?" Again, you should not: we all know that computer systems are highly sensitive and there is no reason to believe that the improvement with O is actually due to O; it may be a result of a biased experimental setup.

This paper demonstrates that measurement bias is significant, commonplace, and unpredictable. Moreover measurement bias is not something that we can just work around: just as with natural and social sciences, we have to take measures to avoid and detect measurement bias.

Acknowledgments

We would like to thank Evelyn Duesterwald, Han Lee, and Andy Georges for valuable feedback.

This work is supported by NSF ITR grant CCR-0085792, CSE-SMA 0509521, IBM, and a ThinkSwiss research scholarship. This material is based upon work supported by the Defense Advanced Research Projects Agency under its Agreement No. HR0011-07-9-0002. The University of Lugano is a member of the HiPEAC Network of Excellence. 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] Alaa R. Alameldeen and David A. Wood. Variability in architectural simulations of multi-threaded workloads. In *IEEE HPCA*,pages 7–18, 2003.

[2] 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.

[3] 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.

[4] Stephen M Blackburn, Perry Cheng, and Kathryn S McKinley. Myths and realities: The performance impact of garbage collection. In *SIGMETRICS*, pages 25–36. ACM Press, 2004.

[5] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *SC*, Dallas, Texas, November 2000.

[6] Amer Diwan, Han Lee, Dirk Grunwald, and Keith Farkas. Energy consumption and garbage collection in low-powered computing. Technical Report CU-CS-930-02, University of Colorado, 1992.

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

[8] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide. http://www.intel.com/products/processor/manuals/. Order number: 253669-027US, July 2008.

[9] John P. A. Ioannidis. Contradicted and initially stronger effects in highly cited clinical research. *The journal of the American Medical Association (JAMA)*, 294:218–228, 2005.

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

[11] 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.

[12] 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.

[13] Han Lee, Daniel von Dincklage, Amer Diwan, and J. Eliot B. Moss. Understanding the behavior of compiler optimizations. *Softw. Pract. Exper.*, 36(8):835–844, 2006.

[14] 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.

[15] 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.

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

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

[18] Dan Tsafrir, Keren Ouaknine, and Dror G. Feitelson. Reducing performance evaluation sensitivity and variability by input shaking. In *MASCOTS*, 2007.