Supporting Experiments in Computer Systems Research

by
Todd Mytkowicz
B.S. Syracuse University, 2001

A thesis submitted to the
Faculty of the Graduate School of the
University of Colorado in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
Department of Computer Science
2010

Author email: Todd.Mytkowicz@colorado.edu

During the past few years, I have had the privilege to work with many people who have given me their friendship, professional advice, and helped me to grow as a researcher.

I would first like to thank my advisor, Amer Diwan. I remember when Amer and I first met to discuss experimental methods in computer systems research; little did I know it at the time, but the kernel of that discussion turned my dissertation topic. Amer has been an ideal advisor, allowing and encouraging me to investigate topics of my own choosing. His unrelenting support of my research and guidance in its dissemination have been instrumental in any success I can claim as a graduate student.

I would like to thank my co-authors, Peter Sweeney and Matthias Hauswirth. Both hosted me for an extended stay at their places of work (Peter at IBM Research and Matthias at the University of Lugano) and for that I am grateful. I appreciate both of their time and effort in what has become my dissertation. Our weekly meetings, and their weekly advice, were helpful in pushing me through some of the difficult issues that arose while completing this work.

Elizabeth Bradley, my co-advisor, I would like to thank for introducing me to the world of dynamical systems. Even before I was her student, Liz took time out of her day to discuss chaos in traditional Artificial Intelligence search problems. Her support and guidance has been crucial in this dissertation: indeed the backbone on which it rests, chaos in computer performance, would have been next to impossible without her guidance.

To my fellow graduate students in the systems lab that have helped me with anything from networking our servers, to working the espresso machine, to walks for espresso elsewhere—I thank you. In particular, I would like to thank those that I work with on a daily basis: Devin Coughlin, Rhonda Hoenigman, Christoph Reichenbach, Dan Knights, John Giacomoni, Zach Alexander, Steven Heck and Amber Roche.

To the programming languages group getting started at CU: I appreciate the time all of you spent listening to my practice talks and providing career advice.

And to Adele Howe and Chuck Anderson: I thank you for your influence on my respect for evaluation in all aspects of experimental work.

Finally, on a more personal note, I would like to thank my family and Emilie Stowell: without their loving support throughout the years I would not have been able to complete graduate school and for this, I thank you.

Abstract

Systems research is an experimental science. Most research in computer systems follows the trend of innovate (e.g. build a novel garbage collector) and then evaluate (e.g. does it significantly speed up our programs). Researchers use experiments to drive their work; they use experiments to identify bottlenecks and then again to determine if their innovations for addressing those bottlenecks are effective. If their experiments are not carried out properly, researchers may draw incorrect conclusions; they may end up wasting time on something that is not really a problem and may conclude their innovations are beneficial even when they are not.

A complicating factor in computer systems experiments is the fact that computer systems are nonlinear dynamical systems, capable of complex and even chaotic behavior. A hallmark of chaos is a sensitive dependence on initial conditions—small changes to the system lead to a large effect on its overall behavior. This sensitivity complicates both *observations* of our systems and *evaluations* of our innovations. It complicates our observations because our measurement tools perturb the system they are observing. It complicates our evaluations because small changes to the environment in which we carry out our experiments can cause large and dramatic changes in system behavior.

In this dissertation, we argue the systems community needs to support experiments with tools that allow a researcher to accurately *observe* her system and methodologies that allow researchers to accurately *evaluate* the impact of their innovations. To support our argument, we introduce two tools that allow researchers to accurately observe their application's behavior and one methodology that allows researchers to accurately evaluate the impact of their innovations.

Contents

Chapter 1

Introduction

Systems research is an experimental science[22; 57; 69]. Most research in computer systems follows the trend of innovate (e.g. build a new whiz bang garbage collector) and then evaluate (e.g. does it significantly speed up our programs). Researchers use experiments to drive their work; they use experiments to identify performance bottlenecks and then again to determine if their innovations for addressing those bottlenecks are effective. If their experiments are not carried out properly, researchers may draw incorrect conclusions; they may end up wasting time on something that is not really a problem and may conclude their innovations are beneficial even when they are not.

Unfortunately, carrying out a proper experiment in computer systems is difficult. As we demonstrate in forthcoming chapters, *common experiments in computer systems research are easily biased, which lead a researcher to incorrect conclusions.*

To support this thesis, we demonstrate that current state of the art measurement tools and evaluation methodologies are biased and thus often produce inaccurate measurements and incorrect evaluations. Bias arises when one experimental setup—or the environment in which we carry out our experiments—favors one particular experimental outcome. For example, we show in a later chapter that the order in which one links object files together overestimate the efficacy of the `gcc` optimization level *O3*.

Bias is a source of inaccuracy in systems experiments and in this dissertation we argue that the systems research community needs to support experiments with tools that allow a researcher to accurately observe her system and methodologies that allow researchers to accurately evaluate the impact of their innovations. Without mitigating the effects of bias on our experiments with these tools and methods, progress in our domain suffers.

1.1 The role of experiments in computer systems research

A major goal of systems research is to build *better* systems—i.e. more energy efficient hardware, smarter operating systems, faster virtual machines, or easier to use software abstractions. To do so requires we experiment. For example, in order to understand how to make our state of the art virtual machine faster, we need to observe its behavior and understand the source of any performance anomalies. In short, experiments drive our work: we experiment in order to understand the behavior of our systems and use that understanding to build better systems. Generally speaking, when we carry out an experiment in systems we follow these three steps:

- `OBSERVATION`: We measure our system in order to find its bottlenecks.
- `INNOVATION`: Once we understand where and how bottlenecks arise in our system, we *hypothesize* an innovation that we hope will fix those bottlenecks.
- `EVALUATION`: We measure our system before and after applying our innovation to test whether our innovation successfully fixes our application's bottlenecks.

Whenever a researcher observes her software in order to find the location of its bottlenecks, she *trusts* her `OBSERVATIONS` are accurate. If they are not, she cannot truly understand the behavior of her system and any innovation she builds based on that faulty understanding is suspect. Likewise, whenever a researcher `EVALUATES` the impact of an innovation on her system's bottlenecks, she *trusts* her evaluation methodology will accurately estimate the impact of her innovation on her system's bottlenecks. If it does not, she may conclude her innovation is useful even when it is not.

Unfortunately, as we show in this dissertation, popular tools for observing our programs behavior are often inaccurate and state of the art methodologies for evaluating our innovations are likely to incorrectly estimate the impact of our innovations. As a consequence, progress in computer systems research suffers.

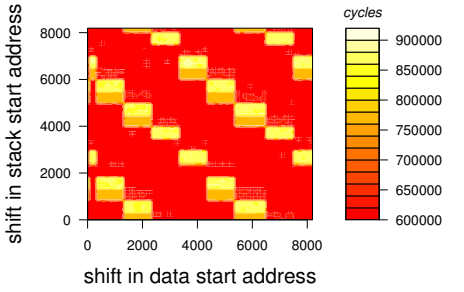
1.2 What makes experiments hard

Complicating both the `OBSERVATION` and `EVALUATION` processes of our experiments is that the behavior of computer systems is *sensitive*. Computer systems are nonlinear dynamical systems, capable of complex and even chaotic behavior[9; 52]. A hallmark of chaos is a sensitive dependence on initial conditions, small changes to the system lead to a large effect on its overall behavior. Ultimately, this sensitivity gives rise to the bias we study in future chapters.

An example here helps make our point: program performance is sensitive to the experimental setup in which we measure performance. We measured the execution time of the simple code in **Figure 1.1** (a) while varying two seemingly innocuous aspects of the experimental setup: the start of the stack section and

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

(a) C code for micro-kernel



(b) Effect of parameters on performance

Figure 1.1: *A micro-kernel that shows extreme sensitivity to its experimental setup*

the start of the data section in an ELF binary, respectively. Figure 1.1 (b) illustrates the effect of these two parameters on the overall runtime of this simple program. The color of a point (x, y) gives the total runtime (in cycles on a Core 2 workstation) when the address of the data segment is at offset x and the start address of the stack is at offset y. We run each configuration 8 times in order to remove the effects of outliers. The color of the point denotes the mean runtime in cycles.

From this simple example we can see that the program's runtime can fluctuate by about 1.5x depending on the location of these two parameters. The fact that program performance is this sensitive to aspects of our experimental setup complicates two parts of the experimental process:

- OBSERVATION: Most of our measurement infrastructures operate from within the systems they are trying to measure. As a consequence, they impact the behavior of the programs they are trying to observe. This perturbation can be large, even at times when we think we have a low overhead measurement infrastructure. As a result, it is difficult to know when we should trust our observations.

- EVALUATION: Subtle and small changes in seemingly innocuous aspects of an experimental setup—or the environment in which we carry out our experiments—can cause large and dramatic changes in overall system behavior. Because different users have different experimental setups, different users can come to *opposite* conclusions about the efficacy of an innovation—even when these evaluations occur on the same machine. This makes it difficult to know when we should trust our evaluations.

In concert, these complications adversely affect the experimental process in computer systems research. The experimental setup in which I run my experiments may be slightly different than your experimental setup and as a consequence, my experiments may not generalize to yours. In other words, my experiments are *biased*.

1.3 Supporting experiments in computer systems

In this dissertation, we acknowledge that the sensitivity of computer systems can bias our observations and our evaluations. With this knowledge, we introduce tools and methodologies that allow researchers to know whether or not the sensitive nature of computer systems behavior is biasing their experiments and if so, how to mitigate its effects.

This dissertation has two parts. In the first two chapters, we demonstrate that common systems experiments are often biased. In the last chapter we show how to mitigate the impact of bias on traditionally expensive metrics through passive observations.

We should note that we do not claim to be exhaustive in this dissertation—that is we do not provide an `EVALUATION` methodology that allows a researcher to trust all of her evaluations. Nor do we provide an `OBSERVATION` tool that is always correct, no matter the measurement. Instead, we pick common tasks that require experimentation in computer systems research and study those.

1.3.1 Trusting our observations

In this chapter we discuss how we can observe our system and trust the result of those observations. In particular, we consider a observation task that is common in computer systems research—profiling. Software researchers profile their programs to find methods that take up a significant amount of overall runtime; a profile guides the researcher to hotspots in her code. Once a researcher knows which methods are hot, she can focus on them to improve her program's performance. If a profile is inaccurate, however, the researcher will waste her time; her efforts will be focused on methods that are not really hot. In this chapter, we demonstrate that existing, state of the art Java profilers (two commercial and two open source products) often produce inaccurate profiles. This work investigates why these profilers are often inaccurate. We use that knowledge to develop a profiler that does not suffer from the same issues as current profilers thus produces more accurate profiles.

1.3.2 Trusting our evaluations

In this chapter we demonstrate that experimental setups—or the environments in which researchers carry out their experiments—are often *biased*[53]. 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. We show bias is commonplace and significant in current computer systems research: it can easily lead to a performance analysis that yields incorrect conclusions. We conclude by evaluating the efficacy of a solution to bias: find its source and use proper randomization techniques to factor out its impact on our conclusions. In this chapter we show not only that bias is pervasive in computer systems evaluation, but also what to do about it.

1.3.3 Mitigating bias through passive observation

If small changes to our programs can have a large impact on their performance, how do we capture measurements that traditionally require a measurement tool to execute a large number of instructions in order to carry out its measurement? In computer systems research, most measurement techniques are *active*; they add expensive instrumentation to a program to capture a metric of interest. In this chapter, we advocate a *passive* approach to measurement. Our idea is simple: we statistically sample the state of a running program and then from those samples, *infer* interesting aspects about the program's behavior—**no instrumentation required**. Specifically, in this chapter we show readily available information during program execution—the height of the call stack and the identity of the current executing function—are good indicators of calling context, or the sequence of function calls that lead up to current executing function[51]. Current active approaches to capturing calling context are expensive, slowing a program by 10% on average. Our passive approach captures these same data without slowing the application. By using a passive approach to measurement, we enable measurements that are not possible with current approaches (e.g. which call paths have the largest number of L1 data cache misses) and at the same time, remove instrumentation as a source of bias.

1.4 Contributions

In concert, these observation tools and evaluation methodology support experiments in computer systems research. While the tools and methodology we introduce in this dissertation do not exhaustively cover all possible types of experiments one would like to do in systems research, we feel they do cover common cases. Moreover, most chapters, in addition to introducing our tools and methodology, also demonstrates the adverse effect of not supporting experimental methods in computer systems research.

The contributions of this dissertation are as follows:

- We demonstrate that compiler evaluations may be biased, leading a researcher to an incorrect conclusion. We show, using a literature survey of 133 recent papers from ASPLOS, PACT, PLDI, and CGO, that prior work does not adequately consider the effects of the bias on their evaluations. We discuss and demonstrate one technique for avoiding bias and one technique for detecting it.

- We demonstrate that current state of the art Java profilers (two open-source and two commercial products) often produce biased and thus inaccurate profiles. In response, we demonstrate how a Java profiler can produce less biased and thus more accurate profiles.

- We introduce a passive approach to collecting call path profiles that has almost no overhead. Current approaches to collecting call path profiles add expensive instrumentation that slows and biases the profiler's results.

Our almost zero overhead approach removes instrumentation as a source of bias in our experiments.

This dissertation is organized as follows: Chapter 2 provides background on the experimental methods we employ in this dissertation. Chapter 3 demonstrates that Java profilers are often inaccurate evaluates introduces a proof of concept Java profiler that is more likely to be accurate. Chapter 4 illustrates the impact of bias on our evaluations and provides a solution for detecting and dealing with bias. Chapter 5 investigates our passive approach to producing call path profiles that allows us to observe aspects of our program's performance that until now have had a prohibitively high overhead. Finally, Chapter 6 puts our work in context of related work and Chapter 7 concludes.

Chapter 2

Experimental methods

In this dissertation we do our best to ensure our own experiments are as free from experimental error as possible. Broadly speaking, experimental error creeps into our OBSERVATIONS and EVALUATIONS through *random effects* and *systematic effects*.

Random effects are fluctuations in our OBSERVATIONS from run to run. They are random because we are just as likely to overestimate the impact of our measurement as we are to underestimate it. Factors in the the environment, such as the operating system scheduling policies or unrelated network and disk activity may affect our measurements as they interact with our system through shared hardware structures (such as caches, TLBs and busses). We in the systems community already have a strategy for removing random effects from our measures: we run experiments on unloaded machines, only access local disks, and use multiple runs along with confidence intervals or error bars to quantify any variation from run to run[11; 34]. In this way, we ensure our OBSERVATIONS, and thus our EVALUATIONS are free from random effects.

In contrast to random effects, systematic effects persist in our experiments, despite multiple runs. Unfortunately, statistics do not give us the tools we need to remove systematic effects from our experiments. Systematic effects arise when our OBSERVATIONS are consistently affected by some aspect of our measurement process. For example, consider a faulty hardware performance monitor that systematically reports 1000 *more* L1 data cache misses than actually happen during a program's execution. We cannot remove this effect from our measurements through statistics. Indeed, many of the phenomena we discuss in later chapters are a result of systematic effects. The tools and methodologies we introduce in this dissertation all try to mitigate the impact of systematic effects on the experimental process.

In this chapter, we generalize our experimental methods. In the sections that follow we describe our approach to measurement, the benchmark programs we use in our experiments and the platforms we carry out our experiments on.

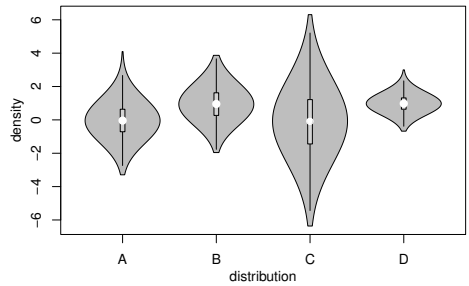


Figure 2.1: *An example violin plot. Each violin is a normal distribution with different mean and variance.*

2.1 Approach to measurement

To mitigate the impact of random effects on our experiments and increase the reproducibility of our results, we follow best practices in regards to our measurements. Specifically:

- We conduct all our experiments on unloaded machines, use only local disks, and repeat each experiment multiple times to ensure that our data are representative and repeatable.
- Where applicable, we conduct our experiments on multiple hardware platforms or Java Virtual Machines (JVMs). This way we ensure our results are not an artifact of one particular machine.
- To minimize measurement overhead, we collect almost all data using hardware performance monitors accessed with PAPI [14].
- Unless otherwise stated, we conduct our experiments in a minimal UNIX environment (i.e., we unset all environment variables that were inessential). Some Linux kernels 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.
- We always use multiple runs and quantify our results with average—or mean—behavior. Unless otherwise noted, we also provide, along with mean behavior, a 95% confidence interval of the mean.

As a consequence of our careful approach to measurement, we generate a large number of observations per experiment. In short, for each experiment we generate a distribution of observations. To aid in our display of these distributions, we employ violin plots[38] (example in **Figure 2.1**). The violin plot is similar to a box plot, except that they also show the probability density of the data at different values. The white dot in each violin gives the median and the thick line through the white dot gives the inter-quartile range (much like a box-plot).

Suite	Benchmark	Description
CINT	gcc	C compiler
	libquantum	Quantum computer simulator
	perlbench	Scripting language interpreter
	bzip	Compression algorithm
	h264ref	Video encoding
	mcf	Single-depot vehicle scheduler
	gobmk	Go program
	hmmer	Computational biology DNA sequence search
	sjeng	Chess program
CFP	sphinx3	Speech recognition
	milc	4D Lattice simulations
	lbm	3D Fluid dynamics
C++	namd	Simulation of biomolecular systems
	dealII	Finite Element Library
	soplex	Solves linear program using Simplex
	povray	Ray tracing program
	astar	2D path finding for game AI
DACAPO	antlr	parser generator
	bloat	Java byte-code optimizer
	chart	plotter and PDF render
	fop	print formatter
	jython	python interpreter
	luindex	text indexing tool
	pmd	Java source code analyzer

Table 2.1: *Benchmarks used in our experiments.*

The width, however, of a violin at y-value y is proportional to the number of times we observed y. In this way, we can succinctly display distributions for our experiments.

2.2 Benchmarks

We use Dacapo[11] and SPEC CPU2006[65] C/C++ benchmark for our experiments. **Table 2.1** gives the suite, benchmark name, and benchmark description for the benchmarks we use in this dissertation. Each Chapter uses a subset of these benchmarks. Here we enumerate how we use each benchmark suite.

2.2.1 SPEC CPU 2006 benchmarks

In Chapter 4 we use all the C benchmarks from CINT (integer) and CFP (floating point) of SPEC CPU2006. We did not consider the non-C programs because the optimization levels would not be comparable between the compilers for dif-

Parameter	Core 2	Pentium 4	m5 (O3CPU)
Operating System	Linux 2.6.25	Linux 2.4.21	NA
Tool Chain	gcc 4.1.3	gcc 4.2.1	gcc 4.1.0
Measurement	papi-3.5.1/pfmon-3.4	papi-3.0.8/perfmon-5.2.16	NA
– toolchain	glibc 2.4-r3	glibc v6	
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
data cache size	4096K	512K	512K
instruction cache size	32K L1, 128K L2	12K L1, 512K L2	128K
TLB entries	512	64	48 Ins. 64 Data

Table 2.2: *Description of the machines used this dissertation.*

ferent programming languages. To generate the data for the figures in this chapter, 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. Even with the train input, it took us 12 days to generate the needed data for our longest running benchmark, `sjeng`.

In Chapter 5 we use the SPEC CPU2006 C and C++ benchmark suite. We were unable to get 4 of the 19 benchmarks working with our passive call path profiling technique: `xalancbmk`, `gcc`, and `gombk` are too large for our instrumentation. Benchmark `omnetpp` uses `setjmp` and `longjmp` for co-routines that our instrumentation is not able to handle. We also had to manually add instrumentation to two methods in `povray` due to a bug in `icc` that causes exits from those methods to not be properly instrumented. We use both the *train* and *ref* input sets in this chapter.

2.2.2 Dacapo benchmarks

In Chapter 3 we use the single threaded Dacapo Java benchmarks with the default inputs. In this chapter we evaluate whether or not four popular state of the art profilers are likely to agree as to the location and magnitude of an application's hotspots. We ignore those Dacapo benchmarks that are multi-threaded (`eclipse`, `lusearch`, `xalan` and `hsqldb`) as this made it difficult to reasonably compare profiles across the different profilers; each profiler handles threads differently which complicates the comparison.

2.3 Platform

We use three platforms in this dissertation: two popular Intel processors and one simulator. We detail our platform in **Table 2.2**. Each chapter uses one of these platforms:

- In Chapter 3, we use the Intel Core 2 workstation for our experiments. To increase the generality of our results, we use two production JVMs:

Sun Hotspot version 1.6.0_12 and IBM J9 version 60. Unless we explicitly say so, we use the Sun JVM for all of our experiments. In both JVM's we always use the default configuration that ships with the JVM (e.g. memory size).

- In Chapter 4, we conduct our experiments on two machines: a Pentium 4 and a Core 2 workstation. On both machines we ran Linux and used PAPI [14] to extract hardware performance monitor information. We added all our instrumentation that accesses PAPI in a wrapper around the main function. The wrapper sets up the data collection before `main` executes and reads out the collected data after `main` exits. Unless we state otherwise, all data in this chapter is for the Core 2 workstation. We report selected data for the Pentium 4 workstation and the m5 simulator using the O3CPU model [10] to demonstrate the generality of our results.

- In Chapter 5, we use the Intel Core 2 workstation for our experiments. Some of our experiments require us to capture the the program counter and the stack pointer of a running program. To accomplish this we made slight modifications to the `pfmon` UNIX tool so as to capture both the *PC* and *SP* registers when we sample with the precise event based sampling (PEBS) mechanism[64]. PEBS allows precise attribution of a certain limited set of hardware events to instructions (e.g. which instruction caused the 1000th L1 data cache miss). In our experiments we sample every one million cycles. In order to interrupt on cycles, we use the PEBS enabled event `instructions_retired` in conjuncture with the `mask` and `inv` parameters of the hardware performance monitor. Specifically, we set `mask` to 8 and `inv` to 1. This has the effect of counting cycles in which 8 or *less* instructions retire per cycle. An Intel Core 2 microprocessor *must* retire anywhere from 0-8 instructions per cycle, so the PEBS counter with these parameters is effectively counting cycles. This is necessary because we need to sample every N cycles and not every N instructions.

Chapter 3

Trusting our OBSERVATIONS

Systems researchers use experiments to drive their work; they use experiments to identify bottlenecks and then again to determine if their ideas for addressing those bottlenecks are effective. If the OBSERVATION aspect of an experiment is biased, a researcher may draw an incorrect conclusion: she may end up wasting her time fixing a bottleneck that is not really a problem.

In this chapter we show that four commonly-used Java profilers (*xprof* [68], *hprof* [67], *jprofile* [27], and *yourkit* [73]) often produce biased OBSERVATIONS. For example, consider a researcher who wants to determine if method $\mathcal{M}$ is hot and thus worthy of optimization. If she uses a profiler that systematically overestimates the time spent in $\mathcal{M}$, she may conclude that $\mathcal{M}$ is hot even when it is not. This phenomenon is called *bias* in other areas of experimental science and in this chapter we show that profiler bias is significant and commonplace: it can easily produce an OBSERVATION that misleads a researcher into optimizing a "hot" method that is not really hot.

This chapter explores the extent of profiler bias by comparing the profiles of four commonly-used Java profilers on the DaCapo benchmark suite. Specifically, we show that these profilers often disagree on both the identity of the hot methods in a program and the time spent in those methods; if two profilers disagree, they cannot both be correct and one or both is producing a biased profile.

A profiler may produce biased profiles for two reasons. First, a profiler's implementation may favor some methods over others. For example, a profiler that ignores native methods (i.e., biased away from native methods) may indicate that the hot methods are in user code when in reality they are all in native code. Second, a profiler may perturb the program as it runs and thus change its profile (i.e. the *observer effect*). This chapter shows that both of these effects contribute to biased profiles.

Generally speaking, we cannot ever know if a profiler produces biased profiles: to know if it is we need a "perfect" profile, or one that lacks any form of

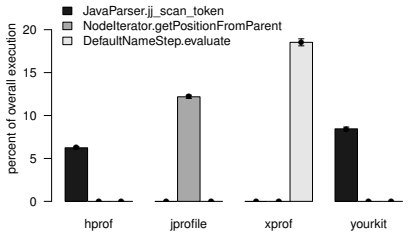


Figure 3.1: *Disagreement in the hottest method for benchmark* `pmd` *across four popular Java profilers.*

bias, that we can use as a baseline. Unfortunately, all profilers operate from within the system they measure and thus suffer from some form of bias. To get around this problem, we introduce the notion of an "actionable" profile. We say a profile is actionable if our acting on a profile yields the outcome we expect. For example, if a profile identifies method $\mathcal{M}$ as contributing 10 seconds to a 20 second program, we *expect* that after speeding up $\mathcal{M}$ by 50%, the program's runtime should drop to 15 seconds. If our expectations are not met, the profiler must have overestimated (or underestimated) the time in $\mathcal{M}$ and was thus not actionable. A nice property of our actionable property is that non-actionable profiles are biased profiles. In other words, we can use the notion of actionable to understand profiler bias.

In this chapter, we evaluate whether four popular profilers produce actionable profiles. In particular we demonstrate how to use causality analysis[58] to determine if a profile is actionable. Causality analysis works by intervention: we change our system (the intervention) and then check if the intervention yields the predicted performance. If the prediction holds, then causality analysis gives us confidence that the profile is actionable; if the prediction does not hold, causality analysis indicates that the profiler is not actionable. Using causality analysis, we demonstrate that the previously mentioned state-of-the-art Java profilers do not produce actionable profiles. A non-actionable profile is a biased profile. To ensure our results are not an artifact of a particular Java virtual machine (JVM) we are using, we show that profilers often produce non-actionable profiles on two different production JVMs (Sun's Hotspot and IBM's J9).

To conclude this chapter, we provide a solution to profiler bias. In particular, we identify the source profiler bias and then introduce a proof-of-concept Java profiler that does not suffer from this source of bias. Finally, we demonstrate that our proof-of-concept profiler produces actionable profiles.

3.1 An example of profiler bias

Figure 3.1 illustrates the amount of time that four popular Java profilers (*hprof*, *jprofile*, *xprof*, and *yourkit*) attribute to three methods from the `pmd`

DaCapo benchmark[11]. There are three bars for each profiler, and each bar gives data for one of the three methods: jj_scan_token, getPositionFromParent, and evaluate. These are the methods that one of the four profilers identified as the hottest method. For a given profiler, $\mathcal{P}$, and method, $\mathcal{M}$, the height of the bar is the percentage of overall execution time spent in $\mathcal{M}$ according to $\mathcal{P}$. The error bars (which are tight enough to be nearly invisible) denote 95% confidence interval of the mean.

Figure 3.1 illustrates that the four profilers disagree dramatically about which method is the hottest method. For example, two of the profilers, *hprof* and *yourkit*, identify the jj_scan_token method as the hottest method; however, the other two profilers indicate that this method is irrelevant to performance, because they attribute 0% execution time to it.

Figure 3.1 also illustrates that even when two profilers agree on the hottest method, they disagree in the percentage of time spent in the method. For example, *hprof* attributes 6.2% of overall execution time to the jj_scan_token method. While *yourkit*, which also identifies this method as hot, attributes 8.5% of overall execution time to this method.

When two profilers disagree, they cannot both be correct—one or more is biased. If a systems researcher uses one of these profilers to understand their system, they may get a biased profile. As a consequence the systems researcher may waste their time optimizing a cold method that will not improve performance. This chapter demonstrates this example is not a corner case but occurs when we profile the majority of our benchmarks.

3.2 Profiler bias is significant and commonplace

In the last section, we showed, at least for one program, four profilers identify three different "hottest" methods. Worse, the hottest method due to one profiler is often cold according to another profiler. Thus, at least some of the profilers produce biased profiles.

In this section we demonstrate that profiler bias is both significant and commonplace. By significant, we mean a profiler frequently overestimates the hotness of a method by 10s of percent. By commonplace, we mean profiler bias shows up with four popular profilers on two production Java virtual machines (Sun Hotspot and IBM J9).

3.2.1 Metrics for quantifying profiler agreement

Before we delve into the extent of profiler bias, we need to define two metrics that allow us to quantify profiler disagreement.

- *Union*$_n$ is the cardinality of the set obtained by unioning the hottest n methods from each profiler. More formally, if the set of n hottest methods according to four profilers are H_1, H_2, H_3, and H_4 respectively, then Union$_n$ is $|H_1 \cup H_2 \cup H_3 \cup H_4|$. In the best case, Union$_n$ is n indicating that the profilers agreed with each other in the identity (but not necessarily

order) of the hottest n methods. In the worst case, Union_n is $n * m$, where m is the number of profilers and $n * m$ indicates systematic disagreement between each of the m profilers(i.e. every profiler disagrees with every other profiler).

- $\textsc{Hotness}_p{}^m$ is the percentage of overall execution time spent executing a method, m according to profiler p. $\textsc{Hotness}_p{}^m$ tells a researcher how much of a benefit they can expect when they optimize m. For example, if the $\textsc{Hotness}_p{}^m$ is 5%, the maximum speedup we expect from optimizing m is about 5%.

We picked the above two metrics because they capture two of the most common ways in which researchers interpret profile information. Specifically, researchers often focus on the few hottest methods especially if these methods each account for more than a certain percentage of program execution time.

To ensure statistically significant results, we always run our experiments multiple times. Consequently, when we report the above two metrics, we report the average behavior across multiple runs (usually 30). For Union_n we order the methods for each profiler based on the average from these multiple runs and then compute the union.

3.2.2 How frequently do profilers disagree?

The agreement of two profiles is a good sign, because the likelihood that two profiles will be biased in the same way is small; therefore, profiler *agreement* builds confidence that both profiles lack bias.

On the other hand, if two profiles disagree then at least one profile must be biased. If disagreement happens only for some corner cases, we may be justified in ignoring it. On the other hand, if profilers disagree frequently, then we cannot ignore it, because that disagreement indicates biased profiles. In this section, we use Union_n and $\textsc{Hotness}_p{}^m$ to illustrate the extent of profiler bias.

Profilers disagree on the hottest method

To understand the extent of profiler bias on the hottest method in a profile, we use the Union_1 metric, across the four profilers, to show that profilers often disagree in the identity of the hottest method (**Figure 3.2**). Each bar in this figure gives the Union_1 metric for one benchmark. Recall that Union_1 will be 1 if all profilers agree on the hottest method and 4 if the four profilers totally disagree on the hottest method.

Figure 3.2 demonstrates profiler disagreement for four of the seven benchmarks (i.e., the bars are higher than 1). In other words, if we use one of these profilers to identify the hottest method in a program, we may end up with a method that is *not* really the hottest method.

We examined the raw data for this figure and saw no obvious trends: e.g., there is no one profiler that always disagrees or agrees with another profiler. For example, *hprof* is just as likely to disagree with *yourkit* as *xprof* on any given

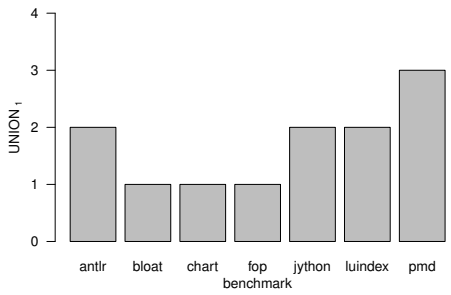


Figure 3.2: *Union*$_1$ *across the four profilers.*

benchmark. As a consequence, we cannot just throw out one profiler and expect that the remaining profilers to all agree with each other.

These results are surprising: the most common usage scenario for profilers is to use them to identify the hottest method; our results show that profilers often fail even in this basic task.

Profilers disagree on the hottest n methods

One explanation for the profiler disagreements in Figure 3.2 discussed in the prior section is that there are other methods that are as hot or nearly as hot as the hottest method. In this case, two profilers may disagree with each other but both could be (almost) correct. For example, if the program spends 20% of its time in the hottest method and 19.9% in the second-hottest method, then some profilers may identify the first method as the hottest while some may identify the second method; both are close enough to being "hottest" that the disagreement does not really matter.

Unfortunately, this is not the case. Consider the profiler disagreement, for example, `pmd` shown in Figure 3.1: *hprof* and *yourkit*, identify the jj_scan_token method as the hottest method; however, the other two profilers indicate that this method is irrelevant to performance, because they attribute 0% execution time to it. In this section we show that this trend holds for the full suite of benchmarks.

Figure 3.3 presents Union$_n$ for n ranging from 1 to 10. A point (x, y) says that Union$_x$ is y. Each point is the mean across the 7 benchmarks and error bars denote 95% confidence interval of the mean. The line $y = 1 * n$ gives the best possible scenario for profiler agreement: with full agreement, the number of unique methods across the top-n hottest methods of the four profilers (i.e., Union$_n$) will be n. The line $4 * n$ gives the worst case scenario for profiler agreement: there is no agreement among the 4 profilers as to what are the top-n hottest methods.

From Figure 3.3 we see that as we consider more methods (i.e., increase the n value), Union$_n$ increases. In other words, even if we look beyond the hottest method and disregard the ordering between the hottest few methods, we still

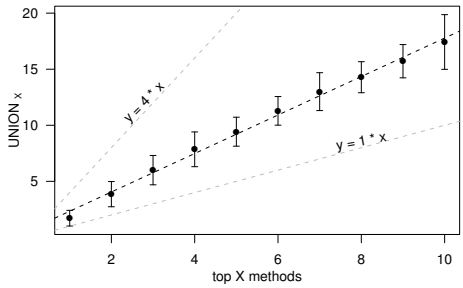


Figure 3.3: *Union$_n$ for our four profilers and n ranging from 1 to 10.*

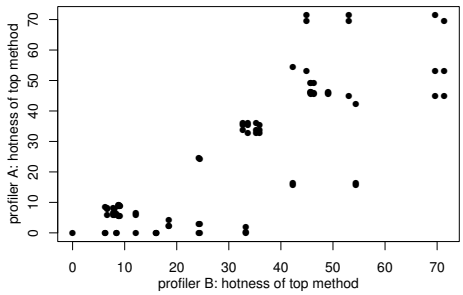


Figure 3.4: *Disagreement in the magnitude of the hottest method for all possible pairs of profilers.*

get profiler disagreement.

3.2.3 By how much do profilers disagree?

The Union$_n$ metric for quantifying disagreement ignores the percentage of time the program spends in the hot methods. This percentage determines whether or not the researcher will even bother to optimize the method; e.g., if the hottest method accounts for only one percent of execution time, it may not be worth optimizing even though it is the hottest method.

In this section we investigate the extent to which profilers disagree as to the hotness of a particular method (**Figure 3.4**). The x-axis in this figure gives HOTNESS$_B{}^m$, or the percentage of overall execution profiler B attributes to method m while the y-axis gives HOTNESS$_A{}^m$, or the overall execution profiler A attributes to method m. For each benchmark, we find the hottest method (m) according to profiler B and then see how much profiler A attributes to method m. We consider all possible pairs of profilers as profilers disagree to identity of the hottest method.

From Figure 3.4, we see that the magnitude of profiler disagreement is large. If profiler A agrees perfectly with profiler B as to the hotness of method m, we should see a straight line at $y = x$. Any deviation from this line shows the

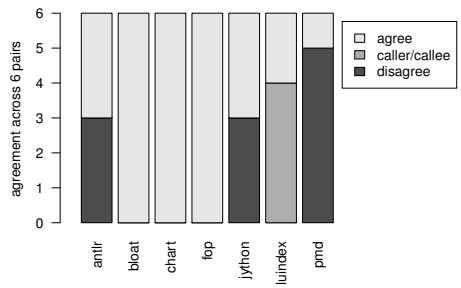


Figure 3.5: *When a pair of profilers disagree, how is that disagreement distributed?*

magnitude of profiler bias. Unfortunately, we see large deviation from the $y = x$ line.

For example, in Figure 3.2 we showed that all four profilers agree as to which method is hottest for benchmark `fop`; however, $\mathcal{P}$yourkit and $\mathcal{P}$jprofile disagree as to the magnitude of this method by 26%, despite $\mathcal{P}$yourkit attributing 70% of overall execution to this method. In many cases, we see that one profiler finds that a method consumes tens of a percent of execution time while another profiler finds that the method consumes little or no execution time. Finally, by inspecting this graph manually, we see no consistent agreement between profilers: e.g. *hprof* sometimes agree and disagrees with *jprofile*.

To summarize, different profilers attribute vastly different amounts of time to the same method. Because researchers use the time spent in a method to decide whether or not they should optimize the method, this disagreement can easily mislead a researcher.

3.2.4 Is profiler disagreement innocuous?

If two profilers identify different methods as the "hottest" but the two methods are in a caller-callee relationship then the disagreement may not be problematic: it is often difficult to understand a method's performance without also considering its callers and callees and thus a researcher will probably end up looking at both methods with both profilers.

Figure 3.5 categorizes profiler disagreement to determine if the caller-callee relationship accounts for most of the disagreement between profilers. A given pair of profilers may (a) agree on the hottest method ("agree"); (b) disagree on the hottest method but the hottest method due to one profiler is a transitive caller or callee of the hottest method due to the other profiler ("caller/callee"); or (c) disagree on the hottest method and the hottest method returned by one profiler does not call (directly or transitively) the hottest method returned by the other profiler ("disagree"). Each bar categorizes the agreement for all profiler pairs for one benchmark; because there are six ($\binom{4}{2}$) possible pairings of four profilers, each bar goes to 6.

From Figure 3.2, we know that all four profilers agree on the hottest method

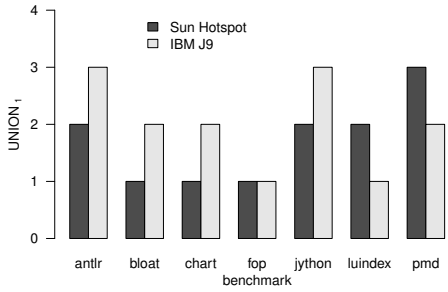


Figure 3.6: *The number of unique methods across suite of profilers.*

for benchmarks `bloat`, `chart`, and `fop`, and thus all the profiler pairs for these benchmarks fall in the "agree" category. However, for the four benchmarks where the profilers disagree, only one benchmark, `luindex` has a caller/callee relationship between the hottest methods identified by the different profilers. Three out of the four times when a pair of profiler's disagree their hottest methods are not in a (transitive) caller/callee relationship.

In summary, profiler disagreement is not innocuous: when two profilers disagree one or both of them may be totally incorrect.

3.2.5 Is the JVM the cause of profiler disagreement?

In Figure 3.2, we quantified profiler agreement on Sun's Hotspot production JVM. We also repeated the same set of experiments using IBM's J9 production JVM. Because J9 does not ship with *xprof*, we used three profilers for J9 instead of four.

For J9 we also found the same kinds of profiler disagreement as with Hotspot (summarized in **Figure 3.6**). For example, across the seven benchmarks, there were only two benchmarks (`fop` and `luindex`) where the three profilers agreed on the hottest method. Thus, profiler disagreement is not an artifact of a particular JVM: we have encountered it on two production JVMs.

3.2.6 Summary

We have shown that profiler bias is (i) significant—four state of the art Java profilers often disagree with each other and (ii) commonplace—occurring for most of our seven benchmarks and in two production Java virtual machines. Because profiler disagreement implies incorrect profiles, this problem is serious: a researcher may waste time and effort optimizing a cold method that has little or no impact on overall program performance.

3.3 Causality analysis

In the prior sections we showed that profiler bias is significant and commonplace. In other words, profilers disagree with each other as to the location and magnitude of a program's hotspots. Profiler disagreement tells us nothing about which profilers are correct. When a pair of profilers disagree with each other, we know nothing about *which* profiler is "correct". If we knew the "correct" profile for a program run, we could evaluate the profiler with respect to this correct profile. Unfortunately, there is no "correct" profile most of the time and thus we cannot definitively determine if a profiler is producing correct results.

For this reason, we relax the notion of "correctness" into "actionable". By saying that a "profile is actionable" we mean that we do not know if the profile is "correct"; however, optimizing the hot methods identified by the profile will yield a measurable benefit. Thus, unlike "correctness" which is an absolute characterization (a profile is either correct or incorrect), actionable is necessarily a fuzzy characterization.

Section 3.5.3 uses the notion of actionable to evaluate profilers. However, this approach is not easy: even if we know which methods are hot, we may not be able to optimize them. Thus, we use a dual of this approach to evaluate profilers: rather than speeding up hot methods, we slow down hot methods (Section 3.3). If a method is hot, then slowing it down further should only make it hotter in the profile. If it does not, then the profile (before or after slowing it down) was not actionable.

In the previous section, we used profiler disagreement to identify that at least one of the profilers generates incorrect profiles. However, profiler disagreement does not tell us if any of the profilers produce actionable profiles. One way to check this is to act on the programs and see if the effects of our actions are reflected in the profiles. Such an analysis is called a causality analysis [58].

Causality analysis, in our context, proceeds in three steps:

Intervene: We transform a method, $\mathcal{M}$, to change the time spent in $\mathcal{M}$. The transformation may take the form of code changes or changes to some parameters that affect the performance of $\mathcal{M}$. For example, we may change the algorithm in a method or ask it to use a different seed for a random number generator.

Profile: We measure the change in execution time due to the intervention for $\mathcal{M}$ and for the entire program. We use a profiler to determine the time spent in $\mathcal{M}$ before and after the intervention. We use a lightweight approach (e.g., the *time* UNIX command) to determine the time spent in the original and intervened program

Validate: If the profiles are actionable, then the change in the execution time for $\mathcal{M}$ should equal the change in the execution time of the program.

There are two significant difficulties with this approach. First, the most obvious intervention is to optimize a hot method. However, it is not always

easy to speed up a method; it may be that the method already uses the most clever approach that we can imagine. This section exploits a key insight to get around this problem: slowing down a method is often easier than speeding up a method. If the profiles are actionable, they should attribute the slow down in the program to the slow down in the method.

Second, the goal of our intervention is to affect the performance of a particular method; however due to memory system effects, our intervention may also affect the performance of other methods [53]. We take two precautions to avoid these unintended effects: (i) In this section, we limit interventions to changes in parameters; thus the memory layout for the method's code before and after the intervention is the same. This ensures that our intervention is not impacting performance due to a change in the program's memory layout, a change we did not intend. (ii) We use interventions that are simple (e.g., contains only computation and minimal memory operations). This ensures that our intervention does not interact with other parts of the program in a way we did not intend.

3.3.1 The intervene and profile steps

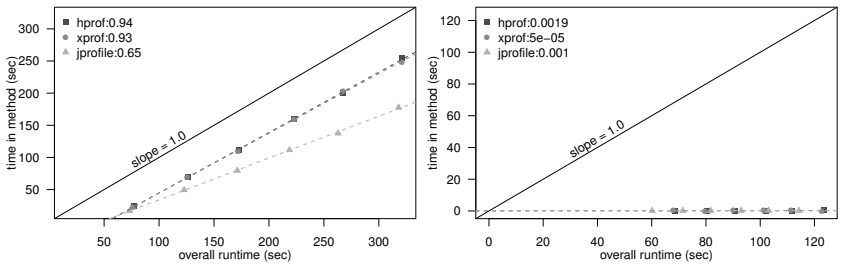
In this section, we used automatic interventions designed to *slow* down a program; in Section 3.5.3 we explore manual interventions designed to *speed* up a program.

We use a Java agent that uses BCEL (a bytecode re-writing library) to inject the intervention code into the program. For the data in this section, we insert a while loop that calculates the sum of the first f Fibonacci numbers, where f is a parameter we specify in a configuration file. We use a Fibonacci computation for two reasons. First, Fibonacci has a small memory footprint and does not do any heap allocation. This simplifies the validation step because we do not need to concern ourselves with memory system effects. Second, we can easily change the amount of slowdown we induce (i.e., the intervention) by altering the value of f. This allows us to change the time spent in the program and see how that change in time is reflected in the profile; specifically, how the profiler reports change in time spent in the method containing the Fibonacci code. Section 3.5 explores the effects of injecting a memory-bound computation in the code.

For each benchmark, we randomly picked two hot methods (from the top-10 hottest methods) for this experiment (second column in **Table 3.1**); these are the methods in which we inject the Fibonacci code. For each experiment, we use the methodology from Section 2 with the exception that we conducted five runs per experiment instead of 30 to keep experimentation time manageable.

3.3.2 The validate step

Figure 3.7 gives the results of our experiments for two methods: the top graph gives data for the ByteBuffer.append method from the `chart` benchmark and the bottom graph gives data for the PyFrame.setlocal method from the `jython` benchmark. There is one set of points for each profiler; the line through the points is a linear fit of the points. We were unable to conduct this experiment



(a) `chart` (b) `jython`

Figure 3.7: *Does application slowdown match slowdown of the method containing the Fibonacci loop.*

for *yourkit* profiler because it does its own bytecode re-writing which conflicts with our method for injecting the Fibonacci code.

The leftmost point is for $f = 100$; each subsequent point adds 200 to f. A point (x, y) on a line for profiler, $\mathcal{P}$, says that when the overall execution time of the program is x seconds, $\mathcal{P}$ attributed y seconds of execution time to the method with the Fibonacci code.

In the perfect case, we expect each profiler's line to be a straight line with a slope of 1.0: i.e., the we expect the increase in execution time for the program to exactly match the increase in the execution time for the method containing the Fibonacci code. The farther a profiler's slope is from 1.0 the less actionable is the profile. To make it easy to see this, we have included an "actionable" line which has a slope of 1.0. In addition, the numbers in the legend give the slope of each line obtained using a linear regression on the data.

For the ByteBuffer.append method from the `chart` benchmark in the top graph, the *hprof* and *xprof* profiler lines have slopes of 0.94 and 0.93, respectively, thus, for this method, *hprof* and *xprof* perform reasonably well. However, *jprofile* has a slope of 0.65 and thus, it does not perform as well.

For the PyFrame.setlocal method from the `jython` benchmark in the bottom graph, all three profiler lines have a slope close to 0, indicating that the profilers do not detect any change in the execution time of the method containing the Fibonacci as we change n! Therefore, none of the three profilers produce actionable data for this method.

Table 3.1 gives the slopes for all benchmarks and profiler pairs. The last row gives the average for each profiler. From this table we see that the slopes are rarely 1: in other words, except for rare cases, *none* of the three profilers produce actionable data.

Benchmark	Method	slope		
		hprof	xprof	jprofile
antlr	CharBuffer.fill	0.45	0.24	-0.05
	CharQueue.elementAt	0.04	0.00	0.11
bloat	PrintWriter.print	0.00	0.13	0.00
	PrintWriter.write	0.42	0.23	0.39
chart	ByteBuffer.append	0.94	0.93	0.65
	ByteBuffer.append_i	0.00	0.00	0.00
fop	PropertyList.findMaker	0.00	0.00	0.00
	PropertyList.findProperty	0.00	0.00	0.01
jython	PyType.fromClass	0.22	0.52	0.55
	PyFrame.setlocal	0.00	0.00	0.00
luindex	jjCheckNAddTwoStates	0.97	1.20	0.99
	StandardTokenizer.next	0.00	0.00	0.00
pmd	NodeIterator.getFirstChild	0.66	0.90	0.82
	JavaParser.jj_scan_token	0.00	0.00	0.00
	mean across methods	**0.26**	**0.28**	**0.23**

Table 3.1: *Slope from the linear regression for Fibonacci injection*

3.4 Understanding the cause of profiler bias

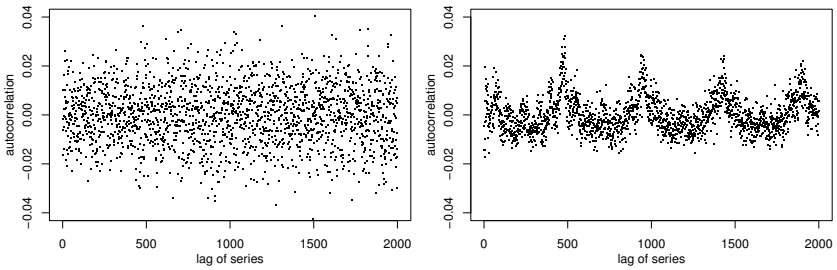
Section 3.2 demonstrates that four state-of-the-art Java profilers often disagree with each other and Section 3.3 demonstrates that the four state-of-the-art Java profilers rarely produce actionable data. This section explores the reason why profilers are producing non-actionable profiles and why profilers produce biased profiles.

3.4.1 The assumption behind sampling for statistically accurate profiles

The four state-of-the-art Java profilers explored in this chapter all use sampling to collect profiles. Profilers commonly use sampling to collect data because of its small overhead. However, for sampling to produce unbiased results, the following two conditions must hold.

First, we must have a large number of samples to get statistically significant results. For example, if a profiler collects only a single sample in the entire program run, the profiler will assign 100% of the program execution time to the code in which it took its sample and 0% to everything else. To ensure that we were not suffering from an inadequate number of samples, we made sure that all of our benchmarks were long running; the shortest benchmark ran for 21.02 seconds, which at a sampling interval of 10ms (which we used), we get about $2,100$ samples.

Second, the profiler should sample *all* points in a program run with *equal* probability. If a profiler does not do so, it will end up with bias in its profile. For



(a) random sampling (b) yieldpoint based sampling

Figure 3.8: *Autocorrelation for (a)* `jython` *using random sampling and (b)* `jython` *using* hprof.

example, let's suppose our profiler can only sample methods that contain calls. This profiler will attribute no execution time to methods that do not contain calls even though they may account for much of the program's execution time.

3.4.2 Do our profilers pick samples randomly?

Because we were careful to satisfy the first condition (by using long runs) we suspected that the profilers were producing non-actionable profiles because they did not satisfy the second condition.

One statically sound method for collecting random samples is to collect a sample at every $t+r$ milliseconds. t is desired sampling interval and r is a random number between $-t$ and t. One might think that sampling every t seconds is enough (i.e., drop the r component) but it is not: specifically, if we sample every t seconds, we would be synchronized with any program or system activity that occurs at regular time intervals [49]. For example, let's suppose that the thread scheduler switches between threads every 10ms. If our sampling interval was also 10ms then we may always take samples immediately after a thread switch. Since performance is often different immediately after a thread switch than at other points (e.g., due to cache and TLB warm-up effects) we would get biased data. The random component, r, guards against such situations.

Figure 3.8 (a) gives the autocorrelation[37] graph for when we take samples using the above approach. Intuitively, autocorrelation determines if there is a correlation between a sequence of sampling intervals at one point in the execution and another point in the execution. More concretely, if the program run produces a sequence, $(x_1, x_2, ..., x_n)$, of sampling intervals, Figure 3.8 (a) plots the correlation of the sequence $(x_1, x_2, ..., x_{n-k})$ with $(x_k, x_{k+1}, ..., x_n)$, for different values of k (k is often called the "lag"). Since correlation produces a value in the range $[-1, 1]$, the autocorrelation graphs also range from -1 to 1 (we have truncated the y-axis range to make the patterns more obvious).

As expected, the autocorrelation graph in Figure 3.8 (a), when we take samples randomly from all points in the program run, looks random. In contrast,

consider the correlation graph for *hprof* (**Figure 3.8** (b)).[1] It exhibits a systematic pattern implying that its sampling intervals at one point in the program run partially predict sampling intervals at a later point; thus the samples are not randomly picked.

In summary, the autocorrelation graph for our profilers look different from the autocorrelation graph for randomly picked samples. Thus, our profilers are not using random samples, which is a requirement for getting correct results from sampling. The remainder of this section explores the cause of this sampling bias.

3.4.3 What makes the samples not random?

To understand why our profilers were not randomly picking samples from the program run, we took a closer look at their implementation. We determined that all four profilers take samples *only* at yield points [2]. More specifically, when a profiler wishes to take a sample, it waits for the program's execution to reach a yield point.

Yield points are a mechanism for supporting quasi-preemptive thread scheduling and garbage collection: they are points in the program where it is safe to switch to a garbage collector or to another thread (e.g., all the GC tables are in a consistent state [23]). As such, compilers often optimize the placement of yield points. For example, as long as the executing code does not allocate memory and does not run for an unbounded amount of time, it is okay to delay garbage collection; thus, a compiler may omit yield points from a loop if it can establish that the loop will not do any allocation and will not run indefinitely. This clearly conflicts with the goal of profilers; in the worst case, the profiler may wish to take a sample in a hot loop but since that loop does not have a yield point, the profiler actually takes a sample sometime after the execution of the loop. Thus, some method other than the one containing the loop may incorrectly get a sample.

Listing 3.1 demonstrates this problem. The `hot` method accounts for most of the execution time of this program and `cold` accounts for almost none of the execution time.[2] Since `hot` does not have any dynamic allocation and runs for a bounded amount of time, the compiler does not put a yield point in it. There is, however, a yield point in `cold` since it contains a call (compilers conservatively assume that a call may eventually lead to memory allocation or recursion). Thus, the `cold` method incorrectly gets all the samples meant for the `hot` method, resulting in a non-actionable profile. Indeed, the *xprof* profiler attributes 99.8% of the execution time to the `cold` method.

In the above example a yield point-based profiler incorrectly attributes a callee's sample to a caller. The problem is actually much worse: JIT compilers aggressively optimize the placement of yield points and unrelated optimizations

[1] The autocorrelation graphs for the other profilers are similar and thus we omit them.

[2] The key thing about the `hot` method is that it is expensive compared to `cold` and does not have calls or loops. We included this code so you can try it out yourself! We created this example from a similar situation we encountered in `antlr`.

```
static int[] array = new int[1024];
public static void hot (int i) {
  int ii = (i + 10 * 100) % array.length;
  int jj = (ii + i / 33) % array.length;
  if (ii < 0) ii = -ii;
  if (jj < 0) jj = -jj;
  array[ii] = array[jj] + 1;
}
public static void cold () {
  for (int i = 0; i < Integer.MAX_VALUE; i++)
    hot(i);
  }
}
```

Listing 3.1: *Code that demonstrates the problem with using yield points for sampling*

(e.g., inlining) may also affect the placement of yield points. Consequently, a profiler may attribute a method's samples to another seemingly-unrelated method.

3.4.4 But why do profilers disagree?

While the above discussion explains why our profilers produce non-actionable profiles, it does not explain why they disagree with each other. If the profilers all use the yield points for sampling, they should all be biased in the same way and thus produce the same non-actionable data. This section shows that different profilers interact *differently* with dynamic optimizations, which results in profiler disagreement.

Any profiler, by its mere presence (e.g. due to its effect on memory layout, or because it launches some background threads), *changes* the behavior of the program (*observer effect*). Because different profilers have different memory requirements and may perform different background activities, the effect on program behavior differs between profilers. Because program behavior affects the virtual machine's dynamic optimization decisions, using a different profiler can lead to differences in the compiled code.

These differences relate to profiler disagreement in two ways: (i) directly, because the presence of different profilers causes differently optimized code, and (ii) indirectly, because the presence of different profilers causes differently placed yield points. While (i) directly affects the performance of the program, (ii) does not affect program performance, but it affects the location of the "probes" which measure that performance. Our results in Section 3.5 suggest that (ii) contributes more significantly to disagreement than (i).

Figure 3.9 illustrates how turning on different profilers change a program's profile generated by *xprof*. The graph in Figure 3.9 has one set of bars for each benchmark and each set has one bar for each of the *hprof*, *jprofile*, and *yourkit* profilers. The height of the bar quantifies the profiler's effect on the

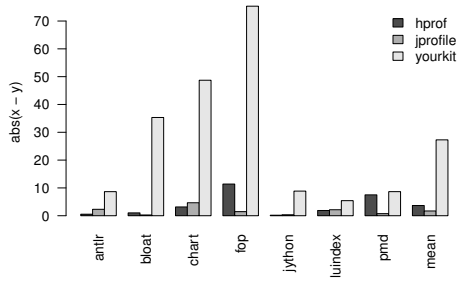


Figure 3.9: *The observer effect due to profilers*

profile observed by *xprof* for the hottest method, $\mathcal{M}$. If *xprof* attributes x% of execution time to $\mathcal{M}$ when no other profiler is running and y% of execution time to $\mathcal{M}$ when a profiler, $\mathcal{P}$, is also running, the $\mathcal{P}$'s bar will have height abs$(x - y)$.

From this graph we see that profilers significantly and differently affect the time spent in the hottest method (according to *xprof*). This observer effect influences where the JIT places yield points. To quantify the observer effect, we used a debug build of Hotspot to count the number of yield points the JIT places in a method. For example, when we profile with *xprof*, the JIT placed 9 yield points per method for the hottest 10 methods of `antlr`, on average. When we used *hprof*, the JIT placed 7 yield points per method.

Although the data in Figure 3.9 illustrates how *xprof*'s profiles change when other profilers simultaneously collect profiles, we see similar behavior when *xprof* is replaced by one of the other profilers.

In summary, the observer effect due to profilers affects optimization decisions which affect the placement of yield points, which in turn results in different biases for different profilers.

3.5 Testing our hypotheses

The previous sections hypothesized that our profilers produce non-actionable profiles because (i) they sample at yield points which biases their profiles and (ii) they interact with compiler optimizations which affects both program performance and the placement of yield points. This section presents results from a proof-of-concept profiler which does not use yield points and shows that this profiler produces actionable profiles.

3.5.1 Implementation

Our proof-of-concept profiler, *tprof*, collects samples randomly using a t of 10ms and r being random numbers between -3 and 3 (Section 3.4.2). *tprof* has two components: (i) a sampling thread that sleeps for the sampling interval

(determined by adding together t and a random number, r, for each sample) and then uses standard UNIX signals to pause the Java application thread and take a sample of the current executing method; and (ii) a JVMTI agent that builds a map of an x86 code address to Java methods so that *tprof* can map the samples back to Java code.

We encountered three challenges in implementing *tprof*.

First, because the JIT may recompile methods and discard previously compiled versions of a method, a single map from Java code to x86 instructions is not enough. Instead, we have different maps at different points in the program execution and the samples also have a timestamp so *tprof* knows which map to use.

Second, because *tprof* operates outside of the JVM, it does not know which method is executing when it samples an interpreted method. Thus, *tprof* attributes all samples of interpreted code into a special "interpreted method". This is a source of inaccuracy in *tprof*; we believe this inaccuracy will be insignificant except for short-running programs which spend much of their time in the interpreter.

Third, Sun Hotspot does not accurately report method locations when inlining is turned on. Thus, we cannot reliably use *tprof* if inlining is enabled.

The latter two limitations are implementation artifacts and not a limitation of profilers that use random sampling.

3.5.2 Evaluating tprof with automatic causality analysis

From Table 3.1 we know that *hprof*, *xprof*, and *jprofile* do not produce actionable profiles; specifically, they do not correctly attribute the increase in program execution time to the increase in the time spent computing the Fibonacci sequence. We now evaluate *tprof* using the same methodology.

Table 3.2 is similar to Table 3.1 except that (i) it includes data for *tprof* along with *hprof*, *jprofile*, and *xprof*; and (ii) we disabled inlining in the JVM when collecting data for this table (Section 3.5.1).

First we notice that *hprof*, *xprof*, and *jprofile* all perform slightly better without inlining than with inlining; Section 3.4.4 explains the reason for this. However, even with inlining disabled, these profilers usually produce non-actionable data.

From the *tprof* column we see that *tprof* performs nearly perfectly: it correctly attributes the increase in program execution time to an increase in the time spent in the method that calculates the Fibonacci sequence.

To increase the generality of our results, we repeated the above experiment, this time injecting a different computation. Specifically, we injected code that allocates an array of 1024 integers, and loops over a computation that adds two randomly selected elements in the array. Once again, we found that other profilers did poorly (with slopes ranging from 0.18 to 0.37) while *tprof* performed nearly perfectly (with slope of 1.02)[3].

[3]We have omitted the full set of results due to space limitations.

Benchmark	Method	slope			
		tprof	hprof	xprof	jprofile
antlr	CharBuffer.fill	1.00	0.78	0.78	0.78
	CharQueue.elementAt	1.00	0.00	0.00	0.00
bloat	PrintWriter.print	1.00	0.01	0.00	0.00
	PrintWriter.write	0.88	0.56	0.23	0.49
chart	ByteBuffer.append	0.99	0.91	0.89	0.65
	ByteBuffer.append_i	0.99	0.00	0.00	0.00
fop	PropertyList.findMaker	0.97	0.00	0.00	0.00
	PropertyList.findProperty	1.00	0.00	0.00	0.01
jython	PyType.fromClass	0.99	0.00	0.00	0.00
	PyFrame.setlocal	0.99	0.00	0.00	0.00
luindex	jjCheckNAddTwoStates	0.99	0.97	0.97	0.98
	StandardTokenizer.next	1.00	0.01	0.01	0.01
pmd	NodeIterator.getFirstChild	1.00	0.79	0.75	0.87
	JavaParser.jj_scan_token	1.00	0.01	0.00	0.00
	mean across methods	**0.99**	**0.29**	**0.26**	**0.27**

Table 3.2: *Slope from the linear regression for Fibonacci injection (no inlining)*

We had posed three hypotheses for explaining non-actionable data from the profilers: (i) reliance on yield points which led to bias (Section 3.4.3), (ii) interactions with optimizations which directly affected profiles (Section 3.4.4), and (iii) interactions with optimizations which affected the placement of yield points and thus bias (Section 3.4.4). Our results indicate that *tprof*, which addresses (i) and (iii) (but not (ii)), performs nearly perfectly.

3.5.3 Evaluating tprof with real case studies

The previous section used causality analysis with synthetic interventions to evaluate the benefit of *tprof*'s sampling strategy compared to *hprof*'s, *xprof*'s, and *jprofile*'s sampling strategy. This section uses realistic interventions instead of the synthetic interventions to make the same comparison.

Speeding up pmd by 52%

tprof reported that java.util.HashMap.transfer was the hottest method accounting for about 20% of overall execution time of pmd. In contrast, *xprof* reported that the method took up *no* execution time and the other profilers (*hprof*, *jprofile*, *yourkit*) reported that three other methods were hotter than this method.

On investigation, we found that pmd creates a HashMap using HashMap's default constructor and then adds many elements to the HashMap. These addition cause the HashMap to repeatedly resize its internal table, each time transferring

the contents from the smaller table to the larger table. Based on *tprof*'s report, we changed the HashMap allocation to use the non-default constructor which pre-allocated 100K entries for the table, thus decreasing the number of times it has to resize the table.

This one line code change sped up the program by 52% with inlining (i.e., the default configuration for the JVM) and 47% without inlining [4]. These performance improvements actually exceed *tprof*'s prediction; we exceeded *tprof*'s predictions because reducing the resizings also reduced the amount of memory allocation, which translated in to better memory system and garbage collector performance.

Speeding up `bloat` by 50%

tprof reported that java.util.AbstractMap.containsValue was the hottest method accounting for 45% of program execution time in `bloat`. The other profilers reported that AbstractMap.containsValue took up 22% of program execution time and reported that other methods were hotter than java.util.AbstractMap .containsValue.

On investigation we found that `bloat` frequently calls AbstractMap .containsValue as part of an assertion. AbstractMap.containsValue does a linear search through the *values* of a map and thus takes time proportional to the number of values in that map. We removed this call by commenting out the assert statement (this does not affect behavior of the program, just the checking of program invariants at run time).

As a result of this change, `bloat` sped up by 50% with inlining and 47% without inlining. *tprof* immediately directed us to this method as the slowest method and even predicted the speedup we got (within 2%). If we had followed the advice of the other profilers, we would still have found this method but not before we had looked at several other methods first.

3.5.4 Summary

Using a combination synthetic and real causality analysis, we have demonstrated that a proof-of-concept profiler, *tprof*, which uses random sampling, produces actionable data.

3.6 Conclusion

What do we do when our program has a performance problem? We use a profiler to find the hot methods in the program and then optimize these methods to speed up the program. If the program does not speed up as predicted by the profile, we typically blame it on a poor interaction with the memory system

[4] We report the performance improvements with both the default JVM configuration and the one with inlining disabled because we collected the profiles with inlining disabled; recall that *tprof* cannot currently handle inlining.

or our lack of understanding of the underlying hardware, but we never blame the profiler. In this chapter, we surprisingly demonstrate that four state-of-the-art Java profilers (*xprof*, *hprof*, *jprofile*, and *yourkit*) often produce incorrect profiles.

We use causality analysis to determine two reasons for why the four profilers produce incorrect profiles. First, the profilers only sample at yield points, a JVM mechanism for supporting quasi-preemptive thread scheduling and garbage collection. Only taking samples at yield points introduces bias into a profile. Second, the profilers perturb the program being optimized (i.e. observer effect) and thus change how the dynamic compiler optimizes the program and places yield points in the optimized code.

Our results are disturbing because they indicate that profiler incorrectness is pervasive – occurring for most of our seven benchmarks and in two production JVM – and significant – all four of the state-of-the-art profilers produce incorrect profiles. Incorrect profiles can easily cause a performance analyst to spend time optimizing cold methods that will have minimal effect on performance. We show that a proof-of-concept profiler that does not use yield points for sampling does not suffer from the above problems.

Chapter 4

Trusting our evaluations

Systems researchers often use experiments to drive their work: they use experiments to identify bottlenecks and then again to determine if their ideas for addressing the bottlenecks are effective. If the EVALUATION of an experiment is biased, 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 idea is beneficial even when it is not.

In this chapter we show experimental setups often bias our EVALUATIONS. For example, consider a researcher who wants to determine if idea I is beneficial for system S. If she measures S and $S + I$ in an experimental setup that favors $S+I$, she may conclude that I is beneficial even when it is not. This phenomenon is called *measurement bias* in the natural and social sciences. This chapter shows that measurement bias is commonplace and significant: it can easily lead to an EVALUATION that yields incorrect conclusions.

This chapter demonstrates measurement bias by exploring if gcc's *O3* optimizations are beneficial for performance (i.e., the I is the optimizations implemented by optimization level *O3*). To explore the effect of measurement bias we consider experimental setups that affect the memory layout of running programs. 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 data; and (ii) link order (the order of .o files that we give to the linker) because it affects code layout. There are numerous ways of affecting the alignment and layout of code and data; we picked two to make the points in this chapter 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) [10]. 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 instead of gcc.

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. Sections 4.3.1 and 4.3.2 discuss potential 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 the 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 [11]. 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 chapter 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 our two techniques directly from techniques in these sciences. The first technique, *experimental setup randomization* (or *setup randomization* for short), runs each experiment in many different experimental setups; these experiments results in a distribution which we summarize using statistical methods to eliminate or reduce measurement bias. The second technique, *causal analysis* [58], establishes confidence that the outcome of the performance analysis is valid even in the presence of measurement bias.

The remainder of the chapter is structured as follows. Section 4.1 explores the origin of measurement bias. Section 4.2 presents our experimental methodology. Section 4.3 shows that measurement bias is significant and commonplace. Section 4.4 demonstrates that measurement bias is unpredictable. Section 4.5 shows that prior work does inadequately address measurement bias. Section 4.6 presents techniques for dealing with bias. Section 4.7 discusses what hardware and software communities can do to help with measurement bias. Finally, Section 4.8 concludes.

4.1 Origin of measurement bias

In Chapter 1.2 and Figure 1.1 we demonstrated that program performance is sensitive to the experimental setup in which we measure performance. an in-

significant 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. If we happen to run our experiments in an experimental setup that has a "pessimistic" ("optimistic") bias to our experiments, our results would look artificially worse (better) than they really are. It is the sensitivity of computer systems that causes measurement bias.

4.2 How measurement bias affects evaluations

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 [43] 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.

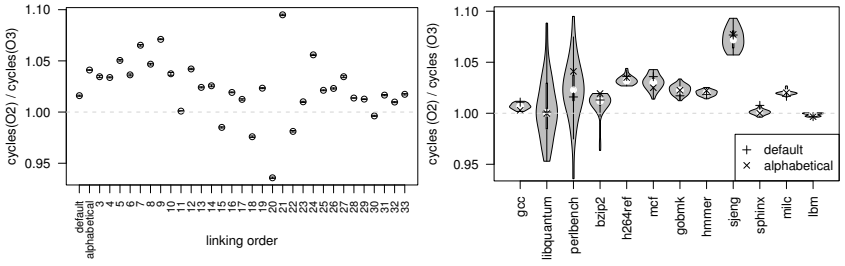
This chapter considers two sources of measurement bias: (i) the UNIX environment size which affects the start address of the stack and thus data alignment; and (ii) link order which affects code layout. We picked these sources because it is well known that program performance is sensitive to memory layout 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 [24]. 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 not the goal of our chapter 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 4.6 to deal with it.

4.3 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



(a) Perlbench (b) All Benchmarks

Figure 4.1: *The effect of link order on the speedup of* O3 *on Core 2.*

optimizations that *O3* introduces (i.e., it does not include optimizations that carry over from *O2*).

4.3.1 Measurement bias due to link order

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

The extent of measurement bias

Figure 4.1 (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 4.1 (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 around the mean.

There are two important points to take away from this graph. First, 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). Second, 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 4.1 (b) uses a violin 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 4.1 (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 "×" points in each violin give the data for the default and alphabetical link orders respectively.

From Figure 4.1 (b) we see that the violins for five benchmarks (`libquantum`, `perlbench`, `bzip2`, `sphinx` and `lbm`) straddle 1.0; thus, for these bench-

marks, 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 even more dramatic: *all* of the violins for the non-FP benchmarks straddle 1.0[1] .

In Figure 4.1 (b) the differences between the maximum and minimum points of a violin are particularly instructive because they give an indication of how much bias one can end up with. On the Core 2 the median difference between the minimum and maximum points is 0.02 while for 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[2]) conclusions about the speedup of *O3* depending on our experimental setup.

We repeated a selection of the above experiments on the m5 simulator using the O3CPU model [10]. 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.

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, branch prediction, etc. 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. We believe, at least for some benchmarks, the link order affects whether or not the loop-stream detector (LSD) on the Core 2 is able to lock a loop into the instruction queue.

On the Core 2 microprocessor, all branches, including taken ones, incur a penalty. To avoid this penalty, the LSD is a logical unit that tries to detect loops and locks them in to a four 16-byte entry instruction queue. Once a loop is locked, it does not incur a penalty on the taken (backward) branch. To test whether link order was interacting with the LSD, we examined the alignment of all loops in the `libquantum` benchmark and found that the alignment of a key loop (in the `toffoli` function) correlates with the performance of the program. Specifically, this loop is 52 bytes in size (and is thus small enough to fit in the 64-byte instruction queue if aligned properly) and `libquantum` performs better if this loop is aligned on a 64-byte boundary than on any other boundary (we tried all multiple-of-eight boundaries up to 64).

[1] Due to time limitations, we did not collect the data for the three SPEC CPU2006 CFP benchmarks on the Pentium 4.

[2] As noted above, the 8% reflects only the SPEC CPU2006 CINT benchmarks.

While we know that the alignment of this key loop correlates with performance and we believe the alignment affects the operation of the LSD, we are unable to conclusively confirm that the LSD is responsible for measurement bias; indeed, there might be other causes. To confirm this explanation we need (i) to know exactly how the LSD operates so we can look at a loop and determine if the LSD would detect it; (ii) more cooperation from the hardware which allows us to collect data that directly measures the behavior of the LSD, and (iii) perhaps even a mechanism for disabling the LSD. Unfortunately we have none and thus we can only provide educated guesses for what, exactly, in the hardware causes a certain performance anomaly.

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.

For example, we used Intel's high-level description of how the LSD works (in Sections 2.1.2.3 and 3.4.2.4 of Intel's Optimization Reference Manual [41]) to write a small loop that should benefit from the LSD. We then unrolled this loop so that it should not benefit from the LSD (the loop became too large and also failed other criteria required by the LSD). We expected the original loop to be significantly faster but instead we found that both loops took the same amount of time. After a full day of exploration where we tweaked both loops we were still unable to come up with an example where one loop conclusively benefited from the LSD and one did not. Clearly, this lack of information about the LSD, specifically, and the hardware, in general, hurts both the users (who cannot get the best performance from their machine investments) and hardware manufacturers (their machines appear slower because compiler writers do not have the knowledge to use hardware features most effectively).

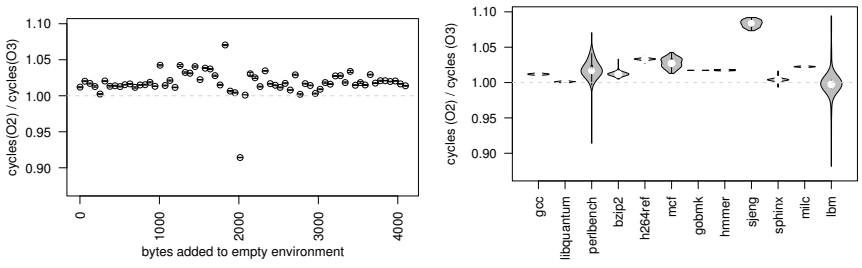
4.3.2 Measurement bias due to UNIX environment size

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

The extent of the measurement bias

Figure 4.2 (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 generated this data using the bash shell We computed each point using five runs each with *O2* and *O3*; the error bars give the 95% confidence intervals around the mean. The tight confidence intervals mean that our runs are easily reproducible. We repeated these experiments with other commonly-used shells and obtained similar results.



(a) Perlbench (b) All Benchmarks

Figure 4.2: *The effect of UNIX environment size on the speedup of* O3 *on Core 2.*

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! Because we repeated the experiment for each data point multiple times, these points are not anomalies but true reproducible behavior.

Figure 4.2 (b) summarizes similar data across all benchmarks. Each violin gives the data for one benchmark and plots all the points for the benchmark (each point corresponds to a particular UNIX environment size). The `perlbench` violin summarizes Figure 4.2 (a). 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 even more dramatic: the violins of six benchmarks straddle 1.0.

In Figure 4.2 (b), the difference between the maximum and minimum points of a violin are particularly instructive because they give an indication of how much of a 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 to obfuscate experimental results [46].

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

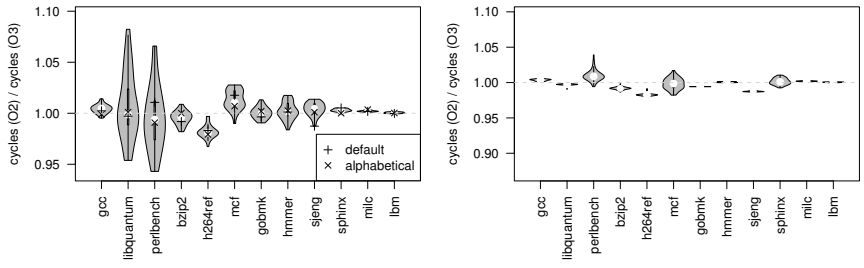
conflicts between loads and stores) depend on alignments of code or data. We verified our explanation by always starting the stack at the same location while changing the UNIX environment size; we got the same *O3* speedup (for all benchmarks except `perlbench`) with different UNIX environment sizes, thus confirming that it was the stack starting location that affected *O3* speedup.

The second reason (which applies only to benchmark `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 stack alignments that lead to the fastest and the slowest execution time. For both of these alignments 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 various reasons, among them loads blocked by preceding stores.

At a high level, we found that the alignment of stack variables was probably causing the measurement bias. At a low level, we found that the LOAD_BLOCK: OVERLAP_STORE was probably causing the 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 PEBS. Unfortunately, PEBS supports only a very limited set of events, and LOAD_BLOCK:OVERLAP_STORE is not part of that set.



(a) Link Order (b) UNIX Environment size

Figure 4.3: *Measurement bias in Intel's C compiler on Core 2.*

4.3.3 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.

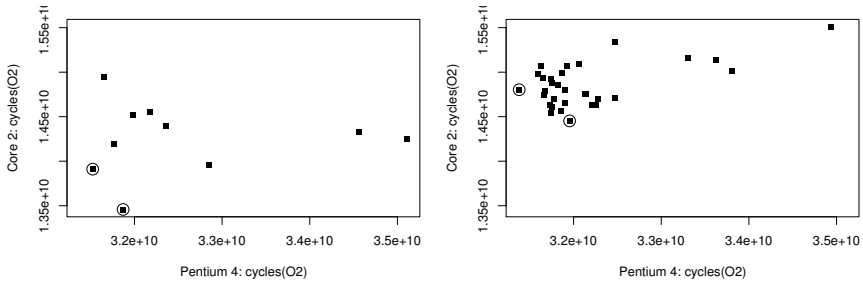
Figure 4.3 (a) presents data similar to Figure 4.1 (b) and **Figure 4.3 (b)** presents data similar to Figure 4.2 (b) except that it uses Intel's C compiler instead of gcc. We see that we get measurement bias also with Intel's C compiler. For our experiments with link order, the violins for 10 (6 for 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 exhibits similar measurement bias than gcc.

4.3.4 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) Environment variables

Figure 4.4: *Measurement bias on Core 2 versus Pentium 4 for* `perlbench`*.*

4.4 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 4.2 (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 4.4** shows for `perlbench` that measurement bias on one machine does not predict measurement bias on another machine. **Figure 4.4 (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 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 4.4 (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.

4.5 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.

4.5.1 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 4.3.1, even simulations suffer from measurement bias.

4.5.2 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 4.3 we show the measurement bias large enough to easily obfuscate a 10% speedup.

4.5.3 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 4.6.1, 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.

4.6 Our 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 evaluating 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.

4.6.1 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.

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 real workloads [11]. As a consequence, the SPEC JVM98 benchmarks may be biased against virtual machines with sophisticated memory managers.

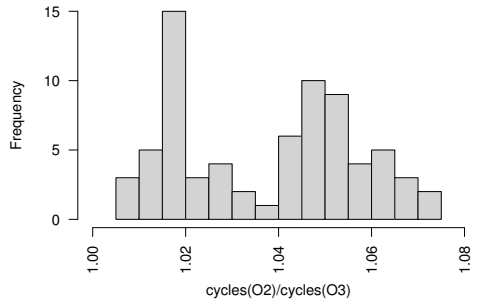


Figure 4.5: *Distribution of speedup due to* O3 *as we change the experimental setup.*

Figure 4.5 evaluates whether or not our benchmark suite (see Table 2.1) 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 4.5 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.

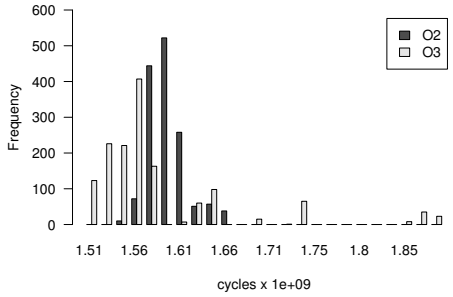


Figure 4.6: *Using variant generation to determine the speedup of* O3 *for* `perlbench`*. At a 95% confidence interval, we estimate the speedup to be* 1.007 ± 0.003*.*

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.

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 4.6 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 (because the ratio is greater than 1).

[3]See Georges *et. al* [34] or most statistics texts for a description of this calculation.

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 [43] reports that as many has 32% of the most high-profile studies 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.

4.6.2 Using causal analysis

Causal analysis is a general technique for determining if we have reached an incorrect conclusion from our data [58]. 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 chapter 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 chapter, 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.

4.6.3 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.

4.7 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.

4.7.1 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 4.6.1 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 [11], go to some length to ensure diversity within the suite; we need more efforts like this for different problem domains and programming languages.

4.7.2 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 many other features of the experimental setup that also cause measurement bias; 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.

4.7.3 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 microkernel 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.

4.7.4 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 [42] lists only nine events that PEBS supports. We hope that in the future hardware will allow us to map all events to specific instructions.

4.8 Conclusion

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 chapter demonstrates that measurement bias is significant, commonplace, and unpredictable: it can severely impact the results of our `EVALUATIONS`. 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. To that end, we introduced two methods to aid systems researchers in their `EVALUATIONS`, both adapted from other areas of experimental science.

Chapter 5

Mitigating bias through passive observation

In the prior two chapters we showed how state of the art tools for observing our systems and methodologies for evaluating the impact of our ideas are often biased. These prior two chapters paint a bleak picture for measuring traditionally expensive measurements from computer systems. An expensive measurement is one that requires the measurement tool to execute a large number of instructions in order for it to carry out its measurement. In this chapter we show how through a passive approach to measurement, we can capture traditionally expensive measurements with very low to no overhead.

In particular, we show how to passively measure call paths and calling context. We call our approach passive because we innocuously observe certain aspects of an executing program and then from those observations, infer traditionally high-overhead characteristics about the program's execution. In this way, we can measure certain aspects of a program's performance that are not possible with traditional call path profilers: such as the number of cache misses per path. Because our `OBSERVATION` tool does not itself produce cache misses, these measurements are not changed by our observations and thus reduces sources of bias. In short, this chapter demonstrates a nearly-zero overhead technique for an important dynamic analysis: collecting call paths and calling contexts. We call our approach Inferred Call Path Profiling, or ICPP for short.

Call path profiles capture the nested sequence of calls encountered at runtime; thus they are useful for determining which sequences of calls consume the most program execution time and for identifying opportunities for aggressive inlining [5; 36] and code specialization [66]. Calling-context profiles are similar to call path profiles except that they produce an abstract value representing each sequence of calls rather than the calling sequence itself. This value is not guaranteed to be unique, but it may be probabilistically so [13]. Calling-context profiles are also useful, e.g., for identifying when a program is executing a call

path that it has not executed before, which may indicate an anomaly [31].

Unfortunately, prior approaches for call path profiling are *active* in that they either require program instrumentation [7; 8; 13; 35; 63; 74] or need to walk the call stack [19; 33; 47] to collect data. Because active profiling requires significant computation during program execution, it may slow down or perturb the program significantly. For example, Zhuang *et al*'s adaptive technique [74] slows down Java programs by an average of approximately 20% and Froyd *et al*'s approach [33] slows down C programs by an average of 7%. If we are collecting additional information at the same time (e.g., data from hardware-performance monitors) this slow-down may unacceptably perturb that information. This chapter describes a *passive* scheme for call path profiling which slows down C and C++ programs by an average of 0.17% (geometric mean) and at most 2.1%.

The key insight behind our approach is that knowing the height of the call stack (in bytes) and the currently executing function uniquely identifies a context most of the time. We call the (stack height, current executing function) pair the *context identifier* since it (usually) identifies a particular context. We show how we can modify the sizes of the activation records so that the context identifier now uniquely identifies a particular context 88% of the time (mean across both C and C++ benchmarks). Finally, we show how to combine the context-identifier with call graph or profile information (from a prior run) to infer the call paths that each context-identifier stands for. Our approach is "passive" since it does not require any additional instructions e.g., to keep track of the context or to traverse the call stack.

We evaluated our approach on C++ and C benchmarks from the SPEC2006 benchmark suites and using two usage scenarios: *offline* and *online*. In the offline scenario we know the inputs of the program in advance so we can do profiling runs in advance of the actual run; these profiling runs help to produce the mapping from context identifiers to call paths. In the online scenario, we do not know the inputs in advance.

We show that our approach uniquely identifies the call path from the context identifier 88% of the time using the offline usage scenario and 74% of the time using the online usage scenario. Moreover, if we are willing to tolerate some ambiguity (i.e., a context identifier possibly maps to more than one call path), our results are even better: for 5-precise (i.e., we map a context identifier to up to five paths, one of which is the correct path) our scheme is correct 98% of the time for the offline scenario and 93% of the time for the online scenario. We show that the run-time cost of our approach is negligible (geometric mean of 0.17% across all benchmarks).

5.1 Motivation

The first line-of-attack when attempting to understand the performance of a program is to measure the end-to-end statistics about the program. For example, we may use UNIX's `time` command to determine how long the program runs for and what fraction of the time it spends in system versus user tasks.

These measurements are cheap and easy to do; however, they provide only a coarse-grained view into the program's performance.

The second line-of-attack is to use tools that measure time spent in each function. If we use sampling (instead of instrumentation) we can do this quite cheaply also: we can use the hardware to trigger interrupts at regular intervals and record the currently executing function at each interrupt. If we sample for a long-enough period, the time spent inside a function will be proportional to the number of samples for that function. Many standard tools, such as UNIX utilities `pfmon`, `gprof`, and the Sun `Hotspot` Java VM use this approach to cheaply collect data. While these measurements are only slightly more expensive than the end-to-end statistics, they are much richer. However, they do not provide any context for a performance analyst to interpret the data. For example, they tell the analyst that function `f` consumes much of the program's execution time but `f` may have many callers; the analyst does not know which call path is primarily responsible for the time spent in `f`.

This chapter shows how we can significantly enrich the above information with *negligible cost*. Specifically, it shows how we can collect not just the time spent in each function but also the *time spent in each call path* using effectively the same data collection mechanisms as the tools above.

5.2 High-level approach

Prior approaches to keeping or capturing calling context all do so *explicitly*—they use instrumentation to gather this information at runtime. For instance, Bond and McKinley propose a technique that explicitly computes calling context by adding instrumentation to each function callsite [13]. In contrast, we show that explicitly computing context at runtime is not necessary—instead we can use readily available information that is a by-product of a program's computation as context. Our technique relies on the fact that calling context is *implicit* in the height of the call stack.

In C and C++, functions store their local variables on the stack, a downward-growing region of contiguous memory that serves as a scratch-pad for data whose lifetime lasts no longer than that of the function invocation. In `x86_64`, the address of the "top" of the stack (often referred as the stack pointer, or SP) is stored in a register dedicated to this use. On every function invocation, the stack pointer is decremented to make room for the callee's *activation record*, which stores parameters, local variables, and other temporary items. When the function returns, it increments the stack back to what it had been before it was called.

For example, if, as in **Figure 5.1**, function A calls function B which then calls C (we will abbreviate this call path as A-B-C), the stack will consist of the activation record for A, followed by that for B, followed by that for C. If A later calls C directly, the stack will contain only the activation records for A and C: the stack pointer will be different if C is called via A-B-C than if it is called via A-C.

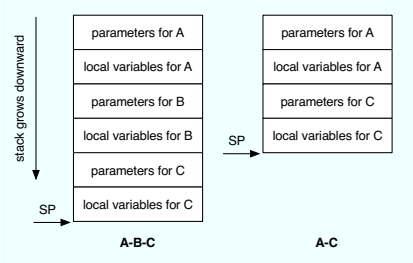


Figure 5.1: *The stack when A calls B calls C and when A calls C directly. The stack pointer in C is different when called via call path A-B-C than when called via call path A-C.*

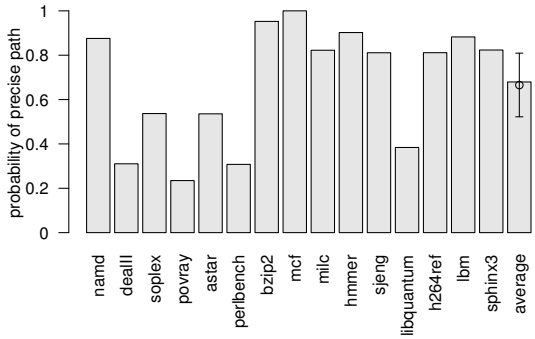


Figure 5.2: *Unaltered C and C++ binaries from the SPEC2006 benchmark suite. The combination of the stack height and the currently executing function uniquely identifies a call path 68% of the time.*

ICPP relies on the hypothesis that the pair—stack height and the identity of the currently executing function—provide a good indicator of a program's calling context. **Figure 5.2** tests this hypothesis using unaltered binaries from the SPEC 2006 C and C++ benchmark suite. There is one bar for each benchmark. The height of a bar gives the fraction of call paths (encountered during a program run) uniquely identified by the stack height and currently executing function. On average, the CID uniquely identifies 68% of call paths. Given this observation, a profile tool can record the stack pointer (SP) and the program counter (PC) in a large number of cases and identify the calling context, rather than add expensive instrumentation or walk the runtime call stack.

The approach we distill in this chapter, ICPP, has four steps:

1 produce a mapping from CIDs to call paths

2 adjust the binary to disambiguate that mapping

3 capture the calling context at runtime

4 process the recorded context identifiers to produce call paths

There are a variety of ways to perform each of these tasks; a client can mix and match different implementations to fit its need for speed and to match its tolerance for ambiguity, time dilation, and other perturbations.

In the following sections we outline several possible implementations of each of the four components of ICPP, describe scenarios under which combinations of these implementations would be useful, and identify situations in which CID is not a good indicator of the calling context and give insight into what we can do about it.

5.3 Step 1: constructing a path map

In order to use context identifiers as a proxy for call paths, we must be able to map a (SP, PC) pair to the call path(s) that lead to it. We have explored constructing this map both statically – analyzing the the program binary and source code – and dynamically, by running an instrumented binary.

5.3.1 Statically constructed path maps

We can statically construct a path map by (1) analyzing the binary to determine how function calls, prologues, and epilogues affect the stack height, (2) constructing a static call graph connecting functions by the caller-callee relationship, and (3) traversing the call graph to generate a list of possible paths and their stack heights.

1. **Binary analysis.** It is relatively straight-forward to analyze a binary to determine how the program changes the stack height: adjustments to the stack are generally limited to call sites, function prologues and function epilogues. However, if a program allocates a dynamically determined

amount of storage on the stack, via the `alloca` call or GCC's variable-length automatic arrays, a static analysis may be unable to determine the affect on the stack.

2. **Constructing the static call graph.** Constructing a complete call graph from a binary is possible, but alias analysis (for determining the targets of function pointers) is less precise at the binary level than with source code [21]. Instead, we used the CIL [56] framework to perform pointer analysis on C source code to determine the targets of function pointers. We did not construct static call graphs for C++, but we could have used Class Hierarchy Analysis [20] analogously to resolve virtual function calls.

3. **Traversing the call graph.** Given a complete call graph, we can traverse it to generate a conservative set of possible call paths. Unfortunately, using this approach the number of possible call paths grows exponentially with the maximum length of a call path.

   If we look up call paths lazily (that is, construct the call paths given a stack height and target function), we can work our way backwards from the target to main, pruning based on call height and shortest paths, although this is still expensive for long call paths. To support this technique without added ambiguity, the CID construction must be invertible. Our CID is invertible, since it uses only addition, but Bond and McKinley's hash-based Probabilistic Calling Context relies on modular arithmetic, so it is not.

In summary, given enough time and space, this static approach can map any CID, even those that may not be executed. However, this technique cannot be applied if any function's activation record size cannot be determined statically.

5.3.2 Dynamically constructed path maps

Dynamically constructing path maps allows us to restrict ourselves to only those paths actually executed. This approach requires offline *training* run(s) that record paths observed at run time along with their context identifiers. We use this information to map context identifiers from later *measurement* runs to their call paths.

We have used `icc`'s `-finstrument-functions` feature, which inserts hooks on each function entrance and exit, to add instrumentation that constructs path maps for C and C++ programs. We use these hooks to build a Calling Context Tree [3] and record the stack height for every call path observed in the running program.

Because function exit hooks are not called when functions are exited via `longjmp`, we use a technique described by Froyd *et al.* [33] that intercepts calls to `longjmp` and uses the stack pointer to determine where in the CCT execution will continue. This technique corrects for `setjmp` / `longjmp` when they are

used for exception handling (as in the SPEC 2006 `perlbench` benchmark), but does not work when these calls are used to implement a coroutine-based threading system, as in the SPEC 2006 `omnetpp` benchmark. We could solve this problem (and support multi-threading in general) by keeping a separate CCT per thread, but since the rest of the SPEC 2006 benchmarks are single-threaded we've chosen not to address it. As it stands this is a current limitation of our approach

Dynamically constructing path maps is efficient because we include in the map only call paths that are actually executed. However, if we conduct separate training and measurement runs to reduce time dilation and other perturbations of the system, we must make sure that the training run covers all the paths executed during the measurement run; otherwise ICPP will report incorrect results.

5.3.3 Summary

We applied the static approach to generating path maps to the SPEC 2006 C benchmarks and found that while it worked on the smaller benchmarks, our first implementation was too slow on `perlbench` and `gcc`. For example, the `perlbench` static call graph consisted of 1,835 nodes and 39,890 edges. A traversal targeting a hot function and stack height found 7326 possible paths and took about 50 minutes. Although it is possible that additional program analysis would allow us to prune enough paths to make this approach feasible, we have instead opted to explore determining path maps dynamically. We present results of the dynamic approach in-depth in Section 5.8.

In summary, statically generating path maps is conservative but computationally expensive and imprecise. Dynamically generating path maps is feasible but may also be imprecise. In Section 5.4 we discuss increasing precision by disambiguating call paths.

5.4 Step 2: binary disambiguation

The mapping from context identifiers to call paths obtained in Section 5.3 may not be *one-to-one*: it is possible that there are several distinct call paths with the same height ending in the same function. In some cases, this ambiguity may be acceptable (e.g. when displaying a hot path to the user, a tool might report two possible hot paths instead) while in others a more precise result may be needed (e.g. when helping a language runtime determine which destructors to call when an exception is thrown).

We now describe several techniques for reducing/eliminating this ambiguity.

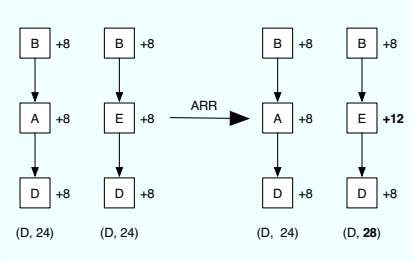


Figure 5.3: *Increasing the size of E's activation record disambiguates B-A-D and B-E-D.*

5.4.1 Activation record resizing

Given a call path F_1-...-F_n (where F_i are functions on the call path), the height of the stack for the call path is:

$$\sum_{i=1}^{n} activation\_record\_size(F_i)$$

By changing the size of an activation record for a function F_i (essentially adding space for unused local variables), we effectively change the stack height for all call paths that include F_i. We use this mechanism to disambiguate the CID to call path mapping.

Figure 5.3 shows two ambiguous call paths, B-A-D and B-E-D. Each node is annotated with size of its activation record. With *Active Record Resizing* (ARR), we can disambiguate these paths by increasing E's activation record size.

Changing a function's activation record size on `x86_64` usually does not require adding any extra instructions: if the program is compiled with a frame pointer (a common occurrence in production code as removing the frame pointer limits debugging) we can simply modify the the immediate operand of the instruction that makes room for the function's local variables on the stack. This modification will, however, change the runtime memory usage of the function.

If a function is lacking a frame pointer we *may* (depending upon the compiler) need to insert a superfluous `sub` instruction in order to affect the size of the activation record. For this reason we always compile our benchmarks with the frame pointer enabled.

This method changes heights on a per function basis, so changing the function's height to disambiguate one call path may cause another path containing that function to become ambiguous. We present an an algorithm to apply ARR globally in Section 5.4.1.

Random search for disambiguation

In this section we describe our search based implementation of ARR disambiguation. We assume that a prior instrumented run of the binary has produced a

set of call paths and their associated stack depths. Disambiguating a set of call paths is a non-trivial global optimization problem. With that in mind, our search process is functional—one could take a more principled approach and add heuristics that take advantage of certain aspects of the search space for a particular problem domain, however we have found a random search to work well for our disambiguation (see Section 5.8 for results).

We repeat the following until a large number of CIDs map to a single call path.

1 We randomly choose two call paths that map to a single CID. To be concrete, we find two call paths that (i) end in the same function and (ii) have the same stack depth.

2 We create a list of those functions that *differ* between the two call paths.

3 We then disambiguate these two call paths by altering the sizes of the activation records of the functions in the list from step (2). In order to speed up the search process and accomplish more disambiguation, with each iteration of this loop, we change the first function in the list's activation record by 16 bytes—and check whether this disambiguates the call path. If it does not, then we alter the second function in the list's activation record by 32 bytes (the third by 48, and so on), always checking if any of these changes disambiguate the two call paths and halting our disambiguation process whenever we find the two paths have been disambiguated. This approach aggressively disambiguates call paths at the expense of runtime stack utilization. Section 5.8.1 discusses this further.

   We always increase the size of the stack by a multiple of 16 because on `x86_64` the value of (SP - 8) must be 16-byte aligned when control is transferred to a function entry point. Future architectures (e.g. the new Intel Core i7) may not have this requirement.

4 If this change increased the total number of CIDs that map to a single call path, we accept the change and go to step 1. Otherwise we undo the change and go back to step 1.

   If after a large number of iterations without forward progress (i.e. any change to disambiguate call paths actually *decreases* the total number of CIDs that map to a single call path), we will accept the change even though it is globally not an optimal choice. This is necessary so as to keep our search from getting stuck in local optima. We used 100 for this parameter.

We repeated these sets of steps until either (i) the total number of CIDs that map to a unique call path was $\geq 97\%$ or (ii) we made no forward progress after 2000 iterations of the loop.

5.4.2 Callsite wrapping

Callsite Wrapping is a disambiguation technique that changes a call path's stack height by surrounding a callsite with decrements and increments to the stack

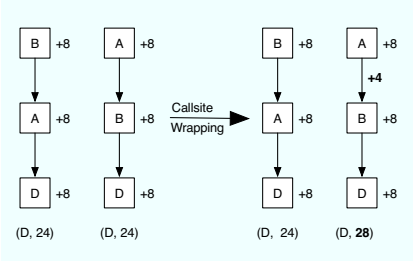


Figure 5.4: *Wrapping the call to B in A disambiguates B-A-D and A-B-D by increasing the stack height along the A-B edge while leaving the B-A edge alone.*

pointer (or equivalently replaces the call at that site with a call to a wrapper function that adds its own activation record to the stack and then calls the original function).

Consider the two call paths in **Figure 5.4**, B-A-D and A-B-D. Because these paths contain exactly the same functions, they will have the same height. ARR is unable to disambiguate these paths: no matter how we resize the activation records of A, B, and D, the sizes of the activation records in these two paths will always add up to identical heights. With Callsite Wrapping, however, we can change the height of the A-B edge while leaving the B-A edge alone.

Callsite Wrapping is more flexible than ARR because it eliminates ambiguity by changing the stack height on a *per callsite*, rather than a per function basis. It also allows us to handle the ambiguity that arises when one function calls another twice: we could wrap one callsite and leave the other alone. But since Callsite Wrapping adds instructions (and possibly new functions), it is likely to be more invasive than ARR.

5.4.3 Function cloning

Function Cloning replaces a call to a function with a call to a copy of that function that contains added disambiguation.

Consider the paths A-B-A-A-D and A-A-B-A-D in **Figure 5.5**. We cannot use Callsite Wrapping (or ARR) to disambiguate these two call paths because both contain exactly the same edges; no matter which callsites we wrap, the total height of these call paths at D will always be the same. If we create a *clone* of function A, A′, that wraps its call to B in order to change the stack height, we then have A-A′-B-A-D and A-B-A-A′-D. The stack height added by A′ calling B is different than that added by A calling B, so these paths now have different heights.

This method of disambiguation requires adding new functions, and requires devirtualization [1] of function pointers and dynamic dispatch, so we consider it to be more invasive than Callsite Wrapping.

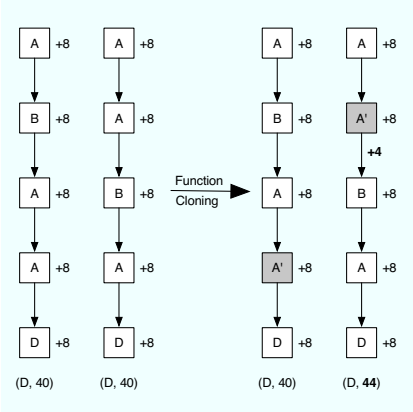


Figure 5.5: *Replacing the call to A in A with a call to a clone, A′, that wraps B will disambiguate A-B-A-A-D and A-A-B-A-D.*

5.4.4 Selective edge instrumentation

Edge instrumentation inserts instructions to keep a record of when one function is called by another. Selectively adding instrumentation along ambiguous paths to keep track of the exact path a program took along the way is a general technique that would allow ICPP to differentiate between any ambiguous call paths at the cost of a large perturbation in the behavior of the program.

5.4.5 Summary

We have presented four techniques for disambiguating call paths: (i) Active Record Resizing, which can be used to distinguish call paths of the same height but with different functions, (ii) Callsite Wrapping, which distinguishes call paths with the same functions but different edges between them, (iii) Function Cloning, which can differentiate between call paths with the same functions and edges but in different orders, and (iv) Selective Edge Instrumentation, which is capable of distinguishing arbitrary call paths, but is expensive.

Bond and McKinley argue that the function for calculating the next context identifier, given a call site and the current identifier, should be commutative, so that it is easy to distinguish between, e.g, call paths B-A-D and A-B-D. We believe that in the general case, the price of commutativity is too high. Our results (cf. Section 5.8.2) show that, for C and C++ programs, addition of the activation record size to the stack pointer (a commutative operation) is good enough to distinguish, on average, 85% of paths in C++ programs and 94% of paths in C programs. Clients with a need for greater precision can apply Callsite Wrapping and Function Cloning to disambiguate further.

In this chapter we have chosen to focus on using Activation Record Resizing in order to demonstrate that ICPP can be precise even without any instrumentation support. The other disambiguation techniques may also be beneficial,

depending on the client of the call path profiling.

5.5 Step 3: capturing calling context at runtime

In order to use ICPP, a tool must capture the stack pointer and the program counter at runtime. It might do so via added instrumentation, by sampling triggered by timer, or with hardware performance monitors.

5.5.1 Sampling with instrumentation

If the client is interested in knowing the context when certain software events occur (such as when a particular API is called, or when a certain error occurs), instrumentation to capture the CID could be added immediately before or after the point of interest. This instrumentation might cause some slowdown, but it would be less than either walking the stack or collecting edge profiles.

5.5.2 Sampling by timer

If the client is interested in observing the call paths executed over period of time, it can use timer-driven statistical sampling to collect this CIDs periodically. The sampling code could be internal, delivered via signals, as is the case with `gprof`, or in a separate program, like `shark`[19], that pauses the targeted program periodically and inspects its state. In either case, the sampler must be able to determine the SP and the PC of the targeted program immediately before the timer was called. Since this sampling involves interrupting the running program, it might cause a large slowdown.

5.5.3 Sampling by hardware performance monitor

Modern processors, such as Intel's Core Duo, have hardware performance monitors with the ability to record the state of registers whenever certain interesting events (such as cache misses) occur. ICPP is ideal for use in this case because the only information it requires to capture the calling context is the state of two registers: SP and PC. This sampling method is minimally invasive because it does not pause the running program.

5.5.4 Summary

Clients of ICPP are able to choose among several different ways of capturing calling context depending on the events for which they require context. Sampling with instrumentation or a timer causes a slowdown and may interfere with other measurements, but affords another opportunity to reduce ambiguity; the sampler instrumentation can look up the CID in the pathmap. If the CID is ambiguous, the instrumentation can walk the stack just enough to disambiguate. In this chapter, however, we have chosen to use a hardware performance monitor, instead of instrumentation, to sample the CID, so we do not walk the stack.

Suite	Benchmark	# Train Paths	# Ref Paths
C++	namd	207	207
	dealII	207317	225313
	soplex	7681	7804
	povray	41495	43425
	astar	547	547
C	perlbench	82441	125312
	bzip2	210	108
	mcf	49	50
	milc	304	304
	hmmer	92	275
	sjeng	16812	18647
	libquantum	336	336
	h264ref	815	3278
	lbm	17	16
	sphinx3	685	703

Table 5.1: *The number of unique call paths for the train and ref inputs.*

5.6 Step 4: producing call paths

Given a context identifier, ICPP must then generally produce a call path (or set of call paths) for a client's consumption.

In previous sections, we have discussed eagerly building a full pathmap, matching CIDs to call paths, and then querying this map. However, another approach is to construct this mapping lazily. That is, we only attempt to discover the paths corresponding to a CID when we are sure that the client will have use for that information. This approach is particularly useful when constructing call paths from a call graph (whether static or dynamic) because doing so is expensive.

ICPP also has a choice in how to deal with ambiguous CIDs. Depending on the needs of the client, it could report all of the possible paths, it could limit itself to the top n paths according to some heuristic (such as frequency of execution), or it could report that it was unable to determine the exact path. In some cases, it may not even be necessary to produce the set of call paths – it might be enough to be able to to compare one context identifier with another.

The exact strategy ICPP uses to report the call paths will depend heavily on how the paths will be used.

5.7 Implementation of ICPP

For these experiments we use the dynamic call path construction discussed in Section 5.3.2. We disambiguate the binaries using Active Record Resizing (cf. Section 5.4.1) and the global optimization search described in Section 5.4.1. We capture the calling context using a hardware performance monitor (cf. Section 5.5.3) and discuss producing paths from an eagerly constructed pathmap as outlined in Section 5.6.

Table 5.1 presents the benchmarks used in our study and the total number of unique call paths for both the `train` and `ref` inputs. The number of paths

varies greatly depending upon the benchmark and input (e.g. `perlbench` has almost 35% more paths in `ref` than in `train`).

5.7.1 Metrics

To evaluate our approach, we categorize call paths as either *precise* or *ambiguous*. We evaluate two usage scenarios of ICPP. In the first scenario, we use the same input to profile and then evaluate. In the second scenario, we profile with one set of inputs and evaluate on a new set.

- In the first scenario, there were two possible outcomes for a given call path: (i) if there are no other paths with the same CID, we call the path "precise"; and (ii) if there are any additional paths with the same CID, we call the path "ambiguous".

- In the second scenario we collect CIDs on the profile input and then categorize call paths on the evaluation input. In addition to "precise" and "ambiguous" there are two additional categorizations for an evaluation call path: (i) if there are some paths in the profile run with the same CID as the evaluation path, but the evaluation path is not among them, we call the path "incorrect"; and (ii) if there are no paths in the profile run with the same CID as the evaluation path, we call the path "missing".

We also classify paths by the total number of call paths that share their CID. We call this number a path's "degree of ambiguity." Under this classification, a path with degree of ambiguity 1 is precise, a path with degrees of ambiguity 2 shares its CID with exactly one other path, and so on.

Note that our definition of a call path—functions that are active on the call stack—does not consider the particular call sites within a function. This definition is the same as some prior work (e.g. `gprof`[35], Spivey[63] and Zhuang et al[74]). If a particular client needs to disambiguate call paths based upon the specific call sites of a function, Section 5.4 lists three techniques that accomplish this goal (Callsite wrapping, Function cloning and Selective edge instrumentation).

5.8 Results

Inferred call path profiling is useful in both an *offline* as well as *online* scenario.

In offline scenarios, we are often running in a controlled environment where we know all the inputs to the program and can run the program multiple times to obtain the information we need. ICPP is invaluable in this case because it enables us to collect context identifiers simultaneously with other data without worrying about perturbing those measurements. Moreover, since we are able to run the program multiple times, we can perform an instrumented run (cf. Section 5.3.2) using the same inputs as the data collection runs in order to translate the context identifiers to full call paths.

Suite	Benchmark	Profile	Lookup	Disambiguation
C++	namd	1.2x	269 cycles	1.2(sec)
	dealII	9.1x	2395 cycles	384(sec)
	soplex	3.6x	497 cycles	16(sec)
	povray	5.8x	721 cycles	960(sec)
	astar	3.4x	331 cycles	2(sec)
C	perlbench	2.9x	1002 cycles	196(sec)
	bzip2	1.4x	320 cycles	< 1(sec)
	mcf	1.6x	208 cycles	< 1(sec)
	milc	1.4x	282 cycles	< 1(sec)
	hmmer	1.0x	230 cycles	3.1(sec)
	sjeng	2.0x	625 cycles	5(sec)
	libquantum	1.1x	270 cycles	2(sec)
	h264ref	2.1x	342 cycles	31(sec)
	lbm	1.1x	163 cycles	< 1(sec)
	sphinx3	1.3x	344 cycles	< 1(sec)

Table 5.2: *The cost of* ICPP *outside program execution.*

In online scenarios, we do not have the luxury of rerunning the program or knowing all the inputs to the program before it runs. ICPP profiling is useful in this setting since context identifiers induce minimal overhead on the running program yet the information is rich enough that if we need the full call path, we can look it up in the path map.

We now evaluate ICPP with respect to the online and offline scenarios.

5.8.1 Cost of ICPP

Broadly speaking ICPP has two kinds of costs.

- **Profile costs:** The cost associated with an offline profile run that gathers data necessary for disambiguation. This is a three phase process each with associated costs: (i) the data collection cost that collects the CID to call path mapping (Section 5.3.2), (ii) the cost of disambiguation of the binary (Section 5.4.1) and (iii) the cost associated with looking up the CID in the call path map (Section 5.6).
- **Measurement costs:** The runtime overhead of running a disambiguated binary (both space overhead and time overhead) while at the same time using the hardware performance monitors to sample the CID at regular intervals (Section 5.5.3).

This section reports on all of these costs.

Cost of ICPP: Profile costs

Table 5.2 details the offline costs of ICPP. In this section We discuss each of these costs in turn.

Cost of offline profiling In order to build the CID to call path mapping, we employ a profile run that maps the CID at each function entry to the current call path

(See Section 5.3.2 for exact details of our approach). In column "Profile" of Table 5.2 we show the overhead of our instrumentation on the `train` inputs. We display overhead as the ratio of the runtime of a program before instrumentation over the runtime of the program with instrumentation. The geometric mean overhead for all of the benchmarks is 2.9x. For some of the programs, the overhead is large (i.e. 14 times slower for `dealII`). We should note that (i) this process is offline and is done once per input for each program and (ii) our instrumentation can be improved upon to reduce the overhead.

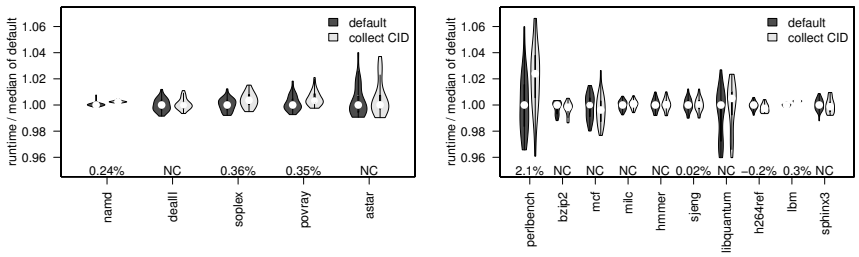
Cost of lookup in context-path map Once we have the path map there is an associated cost of looking up any particular CID in the map. The size of the path maps varies with the benchmarks in our study and thus so too does the cost. In order to investigate the expense of looking up a CID in the path map we wrote a simple tool that loads each of the path-maps into a balanced red-black tree. We selected a large number of random CID values and recorded the average time amount of time it took to look up a particular CID in the map. The "Lookup" column of Table 5.2 details the average number of microprocessor cycles to perform this lookup for each benchmark. The benchmark `dealII` has the largest number of paths (see Table 2.1 for full list) and so it makes sense that it too has the longest lookup times (our red-black tree has $O(\log\ N)$ complexity where N is the total number of entries in the map).

Cost of disambiguation The "Disambiguation" column of Table 5.2 shows the offline runtime cost of disambiguation using our ARR method described in Section 5.4.1. For 12 of the 14 benchmarks the amount of time to disambiguate each benchmark so that either 97% of the call paths uniquely map to a single CID, or the disambiguation process cannot make forward progress, is less than 1 minute. The two that take longer than 1 minute, `dealII` and `perlbench`, take on the order of 6 and 3 minutes, respectively. The reason for this increase in running time is due to the fact that these two benchmarks have the largest number of call paths. We should note that this overhead, however, is a one time cost that could be done, for instance, at the time of the program's installation.

Cost of ICPP: measurement costs

In this section we measure the cost (both in overhead and space) of running a disambiguated binary while at the same time using the hardware performance monitors to sample the CID every one million cycles (see Chapter 2 and Section 2.3 for more details).

Our prior work [53] shows that many aspects of the experimental setup can bias experiments. Concretely, this bias may make our runs look slower or faster compared to a run of the program that does not collect CIDs. To avoid such bias, we ran each experiment (with and without the CID collection) in 32 different experimental setups to obtain a distribution of run times and used t-test to determine if there was a statistically significant difference between the two distributions. We generated the 32 experimental setups by randomly changing



(a) C++ (b) C

Figure 5.6: *Performance impact of recording* CID *for C++ and C programs.*

environment variables and using randomly generated link orders, both of which introduce bias.

Runtime Overhead **Figure 5.6** plots the data for the C++ (a) and C (b) benchmarks. Each violin summarizes the execution times of the runs with ("original") and without ("collect CID") collecting the CID. To enable us to present data for multiple benchmarks on the same violin, we normalized the data to the median of the "original" runs. The white dot in each violin gives the median point and the thick line through the white dot gives the inter-quartile range. The height of the violin gives the range in execution times we observed as a result of changing the experimental setup. The width of the violin gives the distribution of the execution times. The numbers above the x-axis give the mean overhead of "collect CID" as a percentage. NC says that there is no statistically significant slow down (as determined by the t-test). From this data we see that collecting CIDs incurs an insignificant slow down (median of 0 across all benchmarks, geometric mean of 0.17%); it is non-trivial only for `perlbench` at about 2.1%.

Space overhead Active Record Resizing adds to the size of the run-time stack because it disambiguates call paths by increasing the size of function activation records. **Table 5.3** demonstrates that while many programs show little to no increase in stack usage, others, such as `perlbench`, require an increase of more than 2.5x. `perlbench` is extreme in this case because it has many functions with identical activation record sizes called via function pointers from the same callsites. ARR must therefore increase the activation record sizes of these functions in order to distinguish call paths containing them. When disambiguated functions are called recursively, the stack increase can be very large. The average (geometric mean) stack usage increase across all benchmarks, however, is only 1.13x. It is worth noting that (cf. Section 5.8.1) this moderate increase in stack usage has almost no performance impact.

Our current technique for ARR disambiguation balances the time it takes to disambiguate call paths (cf. Section 5.8.1) with the amount of runtime stack utilization. We found that aggressively changing the activation record sizes of

Suite	Benchmark	Increase in Maximum Stack Usage
C++	namd	1.00x
	dealII	1.16x
	soplex	1.05x
	povray	1.13x
	astar	1.04x
C	perlbench	2.54x
	bzip2	1.00x
	mcf	1.00x
	milc	1.02x
	hmmer	1.00x
	sjeng	1.00x
	libquantum	1.61x
	h264ref	1.00x
	lbm	1.00x
	sphinx3	1.00x
Geometric Mean		1.13x

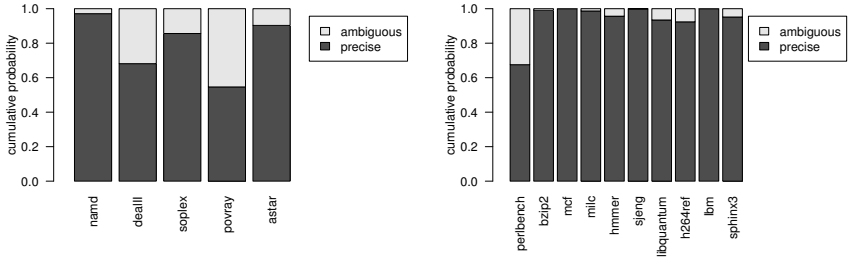
Table 5.3: *The increase in the maximum size of the run-time stack when disambiguating with Active Record Resizing.*

functions in order to disambiguate call paths usually sped up the time it takes to disambiguate a benchmark—however it does so at the cost of runtime stack utilization. Ultimately, our approach was a balance between these two parameters (disambiguation speed and stack usage) and any further implementations of ARR disambiguation are free to choose differently.

5.8.2 Offline scenario

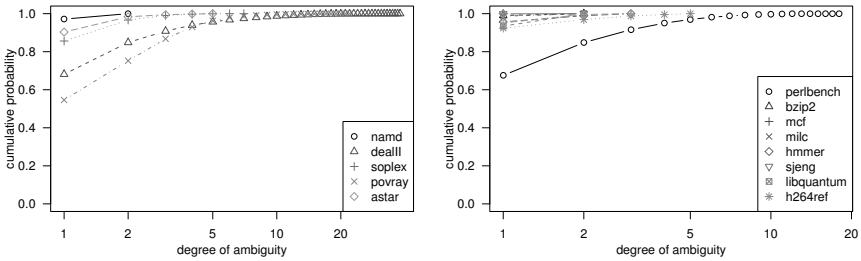
In the offline scenario, we assume that we have all the inputs available in advance and thus we can produce the CID to call path mapping by running the program and observing its behavior (e.g. as in Section 5.9 where we might want to understand which call paths have the highest L1 data cache miss rate).

In Section 5.7.1 we discuss the metrics used in this study. To remind the reader, we briefly reiterate them here. We collected all call paths that we encountered during our program run. There are two possible categorizations for a given call path: (i) there are no other paths with the same CID ("precise"); and (ii) there are additional paths with the same CID ("ambiguous"). Even in the "ambiguous" case, the CID still maps to the correct call path but it maps to additional call paths also. This occurs when there is more than one sequence of calls that leads to a procedure and the different sequences yield exactly the same stack depth. We call the total number of paths that share a path's CID its "degree of ambiguity". A path with degree of ambiguity 1 is precise, a path with degrees of ambiguity 2 shares its CID with exactly one other path, and so on.



(a) C++ (b) C

Figure 5.7: *Precision of path profiles for C++ and C benchmarks (offline scenario).*



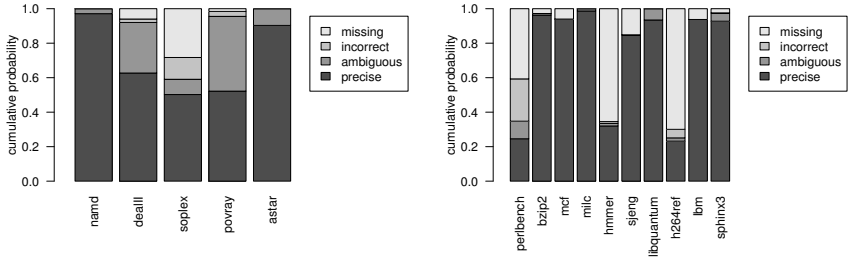
(a) C++ (b) C

Figure 5.8: *Degree of ambiguity (offline scenario).*

We can get ambiguity even after adjusting the sizes of the activation records due to two reasons. First, our algorithm for adjusting activation record sizes is greedy and gives up after a fixed number of iterations, so it may not eliminate all ambiguity. This problem can be reduced by using a smarter algorithm (though we believe, but have not proved, that optimally adjusting activation record sizes is NP-Hard). Second, our ambiguity algorithm does not help the situation when two call paths to a procedure contain exactly the same procedures but in a different order. This problem cannot be fixed by adjusting activation record sizes; Section 5.4.2 describes mechanisms that can eliminate this problem. On manually inspecting the output of our system we found that the second reason was the most prominent one for ambiguity.

Figure 5.7 shows the results for the C++ and C benchmarks. We get a precision of 0.80 for C++ programs (i.e., 80% of the encountered call paths have a CID that maps only to that path) and 0.95 for C programs. One of the C++ programs (`povray`) does suffer from significant ambiguity because many of its call paths contain the exact same functions, but in a different order. Still, our approach yields precise call paths with minimal run-time overhead for the vast majority of the cases.

Figure 5.8 sheds more light into the cases that are ambiguous. It has one



(a) C++ (b) C

Figure 5.9: *Precision of path profiles for C++ and C benchmarks (online scenario).*

line for each benchmark. A point (x, y) says that the probability that the degree of ambiguity of a call path is less than x (that is, that the call path's CID maps to x or fewer call paths) is y. We see that for the C programs, the ambiguity is not serious: even when we have some ambiguity, it is typically no worse than 2. Even for `povray` most (95%) of the time we get an ambiguity of 5 or less. Moreover, if we consider the entire suite of benchmarks, on average 99% of call paths have CIDs that map to 5 or fewer call paths. This is a particularly useful statistic if a client of ICPP is tolerant to some amount of ambiguity.

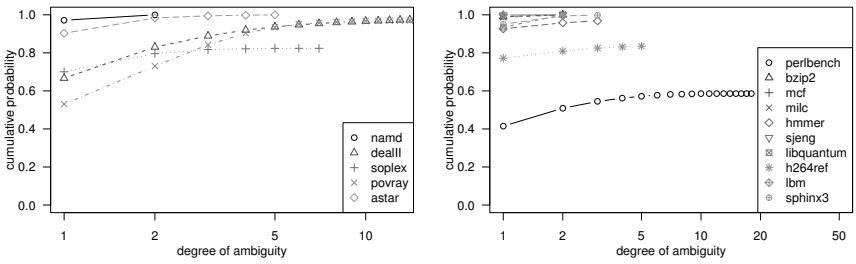
To summarize, our approach produces precise call paths for the offline scenario while incurring minimal overhead for the running program.

5.8.3 Online scenario

In the online scenario, we assume that we do not have have all the inputs available for generating the CID to call path mapping.

In Section 5.7.1 we discuss the metrics used in this study. To remind the reader, we briefly discuss them again here. We ran on a profile input (SPEC `train`) and constructed a mapping from CIDs to call paths. We then collected all call paths from an evaluation input (SPEC `ref`) and categorized these paths. In addition to the "precise" and "ambiguous" outcomes, there are two additional outcomes in the online scenario: (i) there are some paths in the profile run with the same CID as the evaluation path, but the evaluation path is not among them ("incorrect"); and (ii) there are no paths in the profile run with the same CID as the evaluation path ("missing").

Figure 5.9 shows the results for the C++ and C benchmarks. We see that when a call path from the evaluation run is present in the profile run, we are more often than not precise (75% of the time for C++ programs and 73% of the time for C programs). However, we are unable to map the CID for an evaluation path to any profile path 7% of the time for C++ programs and 20% of the time for C programs. These missing paths indicates that our profile run did not exercise the full range of behavior we saw in the evaluation runs. To alleviate this problem, we could either use more inputs for training runs or use a call



(a) C++ (b) C

Figure 5.10: *Degree of ambiguity (online scenario).*

graph analysis to come up with the mapping.

Figure 5.10 is similar to Figure 5.8 and sheds more light into the ambiguous cases. Unlike Figure 5.8, the curves in this graph may not go up to 1.0 because of the missing and incorrect cases. As with Figure 5.8, we see that the curves typically reach their asymptote early; in other words, even when our approach produces ambiguous results, the amount of ambiguity is small—on average 94% of call paths that are not missing or incorrect have CIDs that map to 5 or fewer call paths.

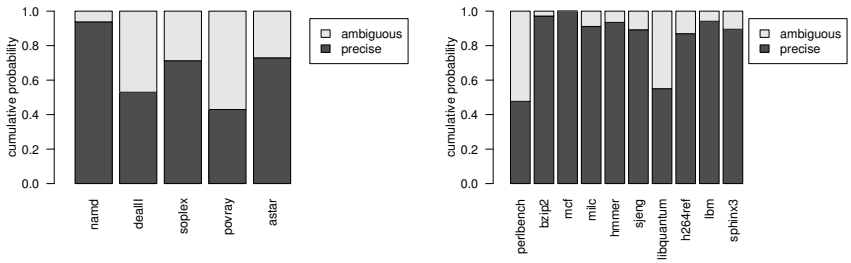
To summarize, while our results are slightly inferior for the online case than the offline case, they are still good: when our system produces call paths, they are correct more than 74.5% of the time. Our system rarely produces incorrect results (4% of the time for C++ programs and 3% of the time for C programs).

5.8.4 Benefit of activation record resizing

So far, all of our results use activation record resizing (Section 5.4.1). **Figures 5.11** and **5.12** give data similar to Figure 5.7 and 5.9 but without activation record resizing. Comparing these graphs, it is clear that activation record resizing greatly increases the number of times we provide a precise path. For our offline scenario, activation record resizing increases our precision from 67% to 80% for C++ programs and from 85% to 95% for C programs. In our online scenario, activation record resizing increases our precision from 60% to 72% for C++ programs and from 67% to 75% for C programs.

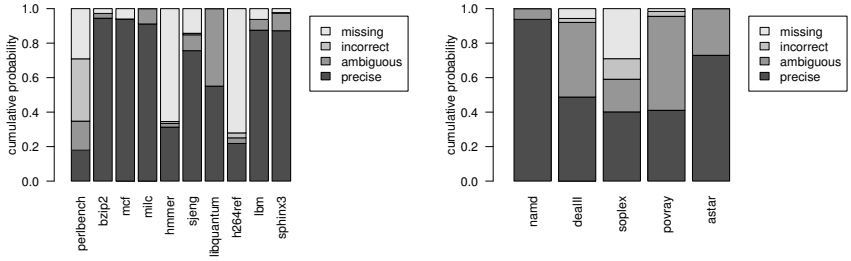
5.9 Usage scenarios

In the prior section we evaluated both the cost and efficacy of Inferred Call Path Profiling in an *offline* and *online* scenario. Now we give some insight into how clients of ICPP could use our technique in their own environments.



(a) offline (b) online

Figure 5.11: *Precision of path profiles for C and C++ benchmarks (offline without activation record sizing).*



(a) offline (b) online

Figure 5.12: *Precision of path profiles for C and C++ benchmarks (online without activation record sizing).*

Hardware-centric call path performance analysis: One compelling use of ICPP is attributing certain hardware events (i.e. L1 data-cache misses, or branch-mispredicts) to the call paths that give rise to the majority of those events. Traditional call path profiling techniques are suspect in this situation because they either add extensive instrumentation (e.g. `gprof`) or walk the stack (e.g. Apple's `shark`)—both of which interfere with the hardware structures (e.g. L1 data-cache) we wish to measure. By splitting our measurement task into two runs, one in which we collect the path-map and then one that collects the CID in hardware performance monitors, we reduce any perturbation in our measurements due to instrumentation.

Debugging support: If the language semantics permit it, a language runtime could use ICPP to determine the call stack when any kind of exception is thrown (e.g. divide by zero, segfault, etc.). If the call path mappings are stored in a cold area of the binary, the language runtime could simply lookup the set of call paths that map to the current CID when an exception is caught. Because an exception denotes anomalous behavior, a bit of ambiguity in the call paths provided to a user is not that much of an issue. Just giving the user a possible set of call paths can greatly aid in their search for a bug. Moreover, because ICPP does not walk the runtime stack aggressive optimizations do not affect its correctness.

Analysis for compilation in a VM: Modern JIT compilers, such as Sun's Hotspot JIT, use a sampling profiler to guide their decisions about which hot methods to optimize. Because these profilers only sample the current executing function, the granularity of their compilation is limited to the method.

If a particular JIT used ICPP for profiling it would have an understanding not just of the hot methods, but also the hot paths—all with negligible amounts of extra instrumentation. This would allow the JIT compiler to increase its granularity of compilation from the method to the call path. Because it is safe to compile multiple paths, this application of ICPP would be tolerant of some amount of ambiguity.

Anomalous behavior detection: Bond and McKinley propose using probabilistic calling context identifiers to detect anomalous (and therefore possibly insecure) call paths in running programs. Our CID could be used for this purpose as well. This usage would be tolerant of some amount of ambiguity because it does not require actually enumerating the possible call paths for a context; it is enough to identify them.

5.9.1 Summary

In this section we have described several possible uses of ICPP, in both online and offline scenarios.

5.10 Conclusion

This chapter introduces a novel approach for capturing calling context and then building call path profiles: Inferred Call-Path Profiling. The key insight behind our technique is that readily available information during program execution—the height of the call stack and the identity of the currently executing function—can uniquely identify the sequence of function calls that lead up to the currently executing function. We call the (stack height, current executing function) pair the *context identifier*. For those instances where the context identifier maps to multiple call paths, we show how to affect the size of function activation records so as to increase the likelihood that any particular context identifier maps to a single call path. Our passive approach to `OBSERVATION` allows us to measure traditionally high-overhead characteristics about the program's execution without the overhead.

We evaluate our approach on the SPEC CPU2006 C and C++ benchmarks in two usage scenarios: *offline* and *online*. In the offline scenario we know the inputs of the program in advance so we can do a profiling run prior to the actual run. In the online scenario, we do not know the inputs in advance. We show that our approach allows the context identifier to uniquely identify 88% of call paths in the offline scenario and 74% of call paths in the online scenario. Because modern processors allow us to periodically sample both the program counter and stack pointer—the constituents of our context identifier—in hardware performance monitors, the overhead of our approach is 0.17% (geometric mean across all benchmarks) and at most 2.1%.

Chapter 6

Related work

In this chapter, we discuss related work. As this dissertation touches on a few disparate areas, we break up the related work into five distinct areas. First, we consider prominent essays that compare experimental science in computer science to other areas of experimental work (e.g. Biology or Physics). Second, we discuss prior work that discusses sources of bias in systems experiments, much in the same vein of our work in Chapter 4. Third, we consider a few papers on the pitfalls and issues in evaluation—or proper experimental methodology—of computer systems experiments. Fourth, we discuss prior work in profiling: both method profiling and call path profiling.

6.1 Computer science is an experimental science

In their joint Turing Award lecture, Simon and Newell, in 1975 talked at length about computer science as an empirical discipline[57]. In their domain, Artificial Intelligence, these two prolific computer scientists argue that computer science is "the study of the phenomena surrounding computers". Their take on computer science research is that it is our job, as computer scientists, to build theories about our domain: much like a Biologist theorizes about the Cell Doctrine or a Geologist theorizes about the movement and flow of plate tectonics. Simon and Newell's view of computer science as an experimental science has been shared by a host of others over the years.

Shortly after the Turning Award lecture of Simon and Newell, Feldman and Sutherland (1979), in an NSF sponsored workshop on rejuvenating experimental computer science argue that universities need to "Use appropriate criteria in evaluating experimental computer science..."[30]. In a CACM article put forth by the ACM Education Board in 1989, Denning et al[18] discuss guidelines for inspiring "general inquiry into the nature of our discipline". They suggest that a major component of research should be in what they call design, but what we here in this dissertation refer to as experimentation.

More recent work, such as that of Feitelson, argues that experimental procedures in computer science research can lead to interesting research results in their own right[29]. Fenton et al argue that computer science, and software engineering in particular, lack rigorous methodologies that aid aid software researchers in evaluating the effectiveness of their creations[32].

These papers all argue that computer science needs more rigor and emphasis on experimental design. Indeed, that is the view that we take in this dissertation. Unlike these prior work, however, we also introduce tools and methodologies that aid researchers in `OBSERVING` and `EVALUATING` their own systems.

6.2 Bias in performance analysis

Even though *bias* is a new term in the computer science literature, systems researchers have long known of its effects on our experiments. For example, Kushman in an unpublished manuscript details what he calls performance anomalies of the UltraSPARC processor[45]. In this work, he demonstrates how removing a single instruction from a kernel can change the performance of a program by a factor of 3x. This work is much like the work in Chapter 4 as we both find sources of bias and work around them in our experimental design.

In their work on OS security, Somayaji et al leverages bias to ensure that all copies of an OS installed on a host of machines do not all suffer the same security vulnerabilities[62]. In this work, the authors use something akin to experimental setup randomization to vary the memory layout of their binary

In [16], Citron demonstrates that using only a few of the programs from SPEC can bias the evaluations of our results. He further shows that many papers (estimates 80%) will not use the entire SPEC suite by 2005. This is not unlike the work of Ziles, in which he shows that using `health` from the Olden benchmark suite [75] to understand the efficacy of optimizations that work on linked data structures can unfairly show these optimizations in a good light.

And, finally in [28], Feitelson illustrates that bias can creep into our experiments in the domain of job scheduling. His work here demonstrates that careful experimental design is paramount to an evaluation that one can trust.

6.3 Experimental methodology in systems research

Prior work in systems has shown methodological issues can significantly impact our results. Papers on proper experimental methodology for different domains of computer systems research have a small role in the literature: for example, [44] informally discusses how to evaluate algorithms experimentally, while Clark et al show how to perform a performance analysis on the virtual machine OS Xen[17].

Blackburn et al find that the size of the heap can bias evaluations of garbage collection[12]. And Georges et al outline proper statistical techniques for a good experimental evaluation of Java programs[34]. This paper is similar to this

prior work in that it develops methodologies and tools that aid programmers in knowing whether or not they can trust their data.

Likewise, Malony et al. have produced a significant body of work on instrumentation perturbation. For example, they study source-level instrumentation that produces traces for SPMD systems [61], and *remove* perturbation by converting a perturbed trace into an approximation of an unperturbed trace. Their approach adjusts the time stamps of each event by subtracting a constant instrumentation overhead α and by taking into account message passing communication between processors. In more recent work [48] they describe how to remove overhead from call profiles generated with source-level instrumentation. They evaluate their approach on profiles of function execution time. While they mention the possibility to collect profiles of other metrics, they do not evaluate their approach on such profiles. As we show in this paper, perturbation in metrics other than cycle or instruction count can be orders of magnitude bigger than the overhead in the number of extra instructions. Moreover, that perturbation generally is not linearly related to the overhead.

Waheed and Rover [71] study three different instrumentation systems for parallel program performance evaluation. While they describe a broad range of aspects of such systems, their view of perturbation is limited to the (time) overhead, in particular to the overhead of flushing trace data buffers in parallel systems. Also in the area of parallel program instrumentation, Hollingsworth and Miller [39] present an approach to dynamically change the amount of instrumentation in the Paradyn system to reduce overhead. Similar to the other papers, they investigate the perturbation in the form of increased execution time.

Najafzadeh and Chaiken [54; 55] investigate the perturbation of hardware performance metrics due to software instrumentation. They propose to validate measurement mechanisms by comparing the resulting models with models generated with different measurement mechanisms, and by running them in different environments. However, they only investigate perturbation of instruction and cycle counts, and only on a limited set of simple benchmarks.

In their work on flow and context sensitive profiling [4], 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 (reading out HPMs every six seconds to prevent counter overflow) to the total metric values produced by their flow and context sensitive profiling system. They show perturbations of metric values by up to a factor of 1442.89. However, for the majority of metrics and benchmarks, perturbation stays below 50%.

Systems researchers commonly address environmental perturbation by using statistical approaches to performance analysis [34]. To compare the performance of multiple variants of a system, a performance analyst executes a program multiple times and uses the distributions of the performance metric values to decide whether the measured improvement is statistically significant. In this paper, instead of determining a statistical symmetrization of performance that overcomes environmental perturbation, we investigate how perturbation changes

depending on the different performance models used.

6.4 Profiling

In this section, we discuss prior work around profiling. We first consider call path profiling tools and then consider method level profiling tools.

6.4.1 Evaluating profiler accuracy

Both industry and the academic literature are full of method profiling tools. Many of these tools use sampling to capture a metric of interest. For example, Arnold and Grove do an offline run to capture the true frequency of caller/callee traversals; they then propose a lightweight approach to approximate this same information[5]. Because the frequency of caller/callee traversals do not change from run to run, they can use their full counts as a baseline to which they compare. Other papers that introduce novel profiling methods use a similar means to validate the accuracy of their profilers (e.g. Shadow Profiling by Moseley et al approximate the frequency of call paths[50] or Duesterwald and Bala's tool for hot path prediction[26]).

Some profiling tools do not sample, but fully instrument code capturing the time spent in each method by wrapping every call to that method[25]. Indeed, both the Netbeans profiler as well as the Eclipse TPTP profiler work in this manor. The overhead for these profilers is considerable; we found about a 1000x application slowdown when every method in an application is instrumented. For this reason, these profilers only instrument those places that a user explicitly specifies—thus reducing overhead to a reasonable level. Their claim is that reduced overhead implies more accurate profiles.

Prior work sometimes uses better results as a proxy for profiler accuracy: the idea being that an accurate profile is more likely to lead to optimizations (e.g. better data layout choices[60]). Buytaert et al [15] found that sampling at yield points to drive dynamic compilation biased the compiler's decisions. As a solution, they use a hardware performance monitor (HPM) driven timer-based approach to sampling. Using HPMs as the foundation for a time-based profiler is similar to our approach with *tprof*; however, a HPM time-based profiler would incur lower overhead. Nevertheless, our *tprof* implementation demonstrates a solution to the problems of sampling at yield points for profiling.

Finally, Whiley uses profiler precision as a proxy for profiler accuracy[72]. His evaluation looks at whether multiple runs of the same profiler agree with each other. A precise profiler (i.e. one that produces the same profile every time it is run) does not mean that the profile is accurate. For example, imagine I have a profiler that always reports foo to take up 100% of program execution, regardless of whether an application even calls foo. This profiler is precise, but not accurate.

6.4.2 Profiling implementations

There is an abundance on prior work that either introduces novel techniques for collecting calling context or directly using calling context (e.g. in optimizations). In the paragraphs that follow we review some of that work.

One of the most popular calling context profilers is the `gprof` tool, which builds caller/callee relationships by instrumentation the epilogue of all functions in a program [35]. These caller/callee pairs are then aggregated—with some loss of precision—into a dynamic call graph for digestion by a user. This work was later extended to more precisely handle programs with mutual recursion and dynamic method binding by Spivey [63]. This type of tool is arguably one of the more useful instruments that a programmer has at her disposal. Inferred Call Path Profiling can provide the same functionality without any of the online instrumentation cost.

Ball and Larus illustrate how to add instrumentation optimally to edges in an intra procedural control-flow graph [7]. Unfortunately this amount of overhead (16% on SPEC95) is large enough to obfuscate certain types of program understanding (e.g. which paths have the largest number of L1 data cache misses) and our focus is on inter procedural call paths.

Obtaining accurate call path profiles usually requires high overhead due to the significant amount of instrumentation. Usually, however, programmers only care about *hot* call paths and can disregard any others. This insight is the basis for most prior work that does selective instrumentation via sampling, bursting or some combination thereof (e.g. [6; 8; 36; 70; 74]).

Bernat and Miller had the insight that a programmer usually only cares about the behavior of a few functions out of the many in an application [8]. Their insight allows a programmer to selectively instrument functions—thus allowing the user to balance instrumentation overhead with accuracy of results.

Zhuang et al introduce an adaptive approach to sampling hot call paths. They walk the runtime call stack and are smart about how far up they walk—i.e. stopping the stack walk when they have hit a part of the calling context tree that they already have sampled [74]. Their overhead is some of the lowest in call path profiling (20% for Java programs).

Walking the runtime call stack is one way to capture calling context—however doing so at every entry to a function boundary is overly obtrusive. Thus, most approaches to stack walking are based upon sampling (e.g. [19; 33; 47]). Froyd et al sample the call stack to produce call paths at an overhead of 7% for the SPEC2000 benchmarks [33]. Likewise, the `Shark` performance tool provides sampled stack walking to identify hot call paths [19]. Our Inferred Call Path Profiling approach could be used as stand-in replacements for both of these approaches.

There are many uses of calling context and call path profiles in prior work (e.g. debugging via stack traces or using call paths to aid in optimization decisions). For instance, Hazelwood and Grove use calling context to aid inlining decisions [36]—as do Arnold and Grove [5]. Likewise, Pettis and Hansen use context to help with code positioning [59]. An interesting area of future work

would be using these techniques with ICPP as a generator of calling context.

Bond and McKinley's Probabilistic Calling Context [13] instruments function epilogues and keeps a hashed value that is built from the prior functions on the call stack and the current executing function. Much like our ICPP, this context information is probabilistic and is likely to provide a unique identifier for context. However, unlike our approach they do not keep track of *which* call paths are on the current runtime stack. Moreover their approach adds instrumentation to compute the hash function at each function entry. Probabilistic Call Context is more likely to provide a unique calling context, however without a significant amount of modification to their technique it is unable map their context identifier to call paths.

Our ICPP approach to profiling is similar Shye et al's path profiling technique. In this work, the authors use the branch target history buffer of modern CPUs to extract the path of execution.

Finally, the key insight behind ICPP was independently developed by Inoue and Nakatani[40] and presented at the same conference. Their technique for capturing calling context works much like our ICPP approach, however, their approach only captures a few levels of context up the calling context tree, and us thus less precise than our approach.

Chapter 7

Conclusion

In this dissertation we have argued that the computer systems research community needs to support experimentation with tools that allow a researcher to accurately *observe* her system and methodologies that allow researchers to accurately *evaluate* the impact of their innovations. In particular we have given two concrete examples as to *why* we think the computer systems community should support experimentation.

1. We have shown how current state of the art methodologies for `EVALUATING` the efficacy of a compiler optimization may be biased. Further, we have shown that the extent of bias is pervasive, occurring on multiple machines and even in one simulator. Because of this bias, a researcher can easily be lead to an incorrect conclusion.

2. We have shown that four popular, state of the art Java profiling tools for `OBSERVING` Java application performance are likely to produce inaccurate profiles. Because of this inaccuracy, a researcher can easily be lead astray and end up spending time and effort optimizing a method that is not really hot.

We acknowledge that the sensitivity of computer systems can both impact our observations and our evaluations. In turn, we introduce two tools for observing our system, both of which allow researchers to understand the performance of their system. We have introduced a methodology that allows researchers to know whether or not the sensitive nature of computer systems behavior is impacting their experiments.

Bibliography

[1] Gerald Aigner and Urs Hölzle. Eliminating virtual function calls in C++ programs. In *ECOOP '96: Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 142–166, London, UK, 1996. Springer-Verlag.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[3] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

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

[5] Matthew Arnold and David Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:51–62, 2005.

[6] Matthew Arnold and David Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 51–62, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[8] Andrew R. Bernat and Barton P. Miller. Incremental call-path profiling: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(11):1533–1547, 2007.

[9] Hugues Berry, Daniel Gracia Perez, and Olivier Temam. Chaos in computer performance. *Chaos*, 16(1):013110, 2006.

[10] 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.

[11] 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 '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, New York, NY, USA, October 2006. ACM Press.

[12] Stephen M Blackburn, Perry Cheng, and Kathryn S Mckinley. Myths and realities: The performance impact of garbage collection. In *In Proceedings of the ACM Conference on Measurement and Modeling Computer Systems*, pages 25–36. ACM Press, 2004.

[13] Michael D. Bond and Kathryn S. McKinley. Probabilistic Calling Context. *SIGPLAN Not.*, 42(10):97–112, 2007.

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

[15] Dries Buytaert, Andy Georges, Michael Hind, Matthew Arnold, Lieven Eeckhout, and Koen De Bosschere. Using hpm-sampling to drive dynamic compilation. In *OOPSLA '07: Proceedings of the 22st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (Montreal, Canada)*, pages 553–568, New York, NY, USA, Oct. 2007. ACM.

[16] Daniel Citron. Misspeculation: partial and misleading use of spec cpu2000 in computer architecture conferences. *SIGARCH Comput. Archit. News*, 31(2):52–61, 2003.

[17] Bryan Clark, Todd Deshane, Eli Dow, Stephen Evanchik, Matthew Finlayson, Jason Herne, and Jeanna Neefe Matthews. Xen and the art of repeated research. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 47–47, Berkeley, CA, USA, 2004. USENIX Association.

[18] D. E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, 1989.

[19] Apple Computer. Shark. http://developer.apple.com/performance.

[20] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, London, UK, 1995. Springer-Verlag.

[21] Saumya Debray and Robert Muth. Alias analysis of executable code. In *In POPL*, pages 12–24, 1998.

[22] Peter J. Denning. Is computer science science? *Commun. ACM*, 48(4):27–31, 2005.

[23] A. Diwan, E. Moss, and R. Hudson. Compiler support for garbage collection in a statically typed language. *SIGPLAN Not.*, 27(7):273–282, 1992.

[24] 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.

[25] M. Dmitriev. Selective profiling of java applications using dynamic bytecode instrumentation. In *ISPASS '04: Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 141–150, Washington, DC, USA, 2004. IEEE Computer Society.

[26] Evelyn Duesterwald and Vasanth Bala. Software profiling for hot path prediction: less is more. *SIGPLAN Not.*, 35(11):202–211, 2000.

[27] Ej technologies: Commercial java profiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[28] Dror G. Feitelson. Experimental analysis of the root causes of performance evaluation results: A backfilling case study. *IEEE Trans. Parallel Distrib. Syst.*, 16(2):175–182, 2005.

[29] Dror G. Feitelson. Experimental computer science: The need for a cultural change. Unpublished Manuscript, December 2006.

[30] Jerome A. Feldman and William R. Sutherland. Rejuvenating experimental computer science: a report to the national science foundation and others. *Commun. ACM*, 22(9):497–502, 1979.

[31] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *SP'03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[32] N. Fenton, S.L. Pfleeger, and R.L. Glass. Science and substance: a challenge to software engineers. *Software, IEEE*, 11(4):86–95, Jul 1994.

[33] Nathan Froyd, John Mellor-Crummey, and Rob Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 81–90, Cambridge, Massachusetts, 2005. ACM.

[34] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *OOPSLA '07: Proceedings of the 22nd ACM SIGPLAN conference on Object-oriented programming, systems, languages and applications*, 2007.

[35] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, Boston, Massachusetts, United States, 1982. ACM.

[36] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, pages 253–264, San Francisco, California, 2003. IEEE Computer Society.

[37] R. Hegger, H. Kantz, and T. Schreiber. Practical implementation of nonlinear time series methods: The TISEAN package. *Chaos*, 9(2):413–435, 1999.

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

[39] Jeffrey K. Hollingsworth and Barton P. Miller. An adaptive cost system for parallel program instrumentation. In *Euro-Par '96: Proceedings of the 2nd International Euro-Par Conference*, pages 88–97, 1996.

[40] Hiroshi Inoue and Toshio Nakatani. How a java vm can get more from a hardware performance monitor. *SIGPLAN Not.*, 44(10):137–154, 2009.

[41] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/products/processor/manuals/. Order number: 248966-016, November 1997.

[42] 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.

[43] 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.

[44] D. Johnson. A theoretician's guide to the experimental analysis of algorithms, 1996.

[45] N. Kushman. Performance nonmonotonicities: A case study of the ultrasparc processor. Technical report, Cambridge, MA, USA, 1998.

[46] 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.

[47] Allen D. Malony, Sameer Shende, Robert Bell, Kai Li, Li Li, and Nick Trebon. Advances in the TAU performance system. *Performance analysis and grid computing*, pages 129–144, 2004.

[48] Allen D. Malony and Sameer S. Shende. Overhead compensation in performance profiling. In *Euro-Par '04: Proceedings of the 10th International Euro-Par Conference*, volume 3149/2004 of *Lecture Notes in Computer Science*, pages 119–132. Springer-Verlag, 2004.

[49] Steven Mccanne and Chris Torek. A randomized sampling clock for cpu utilization estimation and code profiling. In *In Proc. Winter 1993 USENIX Conference*, pages 387–394. USENIX Association, 1993.

[50] Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. Shadow profiling: Hiding instrumentation costs with parallelism. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 198–208, Washington, DC, USA, 2007. IEEE Computer Society.

[51] Todd Mytkowicz, Devin Coughlin, and Amer Diwan. Inferred call path profiling. *SIGPLAN Not.*, 44(10):175–190, 2009.

[52] Todd Mytkowicz, Amer Diwan, and Elizabeth Bradley. Computer systems are dynamical systems. *CHAOS*, 2009.

[53] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 265–276, New York, NY, USA, 2009. ACM.

[54] Haleh Najafzadeh and Seth Chaiken. Validated observation and reporting of microscopic performance using pentium ii counter facilities. In *WOSP '04: Proceedings of the 4th international workshop on Software and performance*, pages 161–165, New York, NY, USA, 2004. ACM Press.

[55] Haleh Najafzadeh and Seth Chaiken. Towards a framework for source code instrumentation measurement validation. In *WOSP '05: Proceedings of the 5th international workshop on Software and performance*, pages 123–130, New York, NY, USA, 2005. ACM Press.

[56] 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 '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[57] Allen Newell and Herbert A. Simon. Computer science as empirical inquiry: symbols and search. *Commun. ACM*, 19(3):113–126, 1976.

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

[59] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, 1990.

[60] Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. An efficient profile-analysis framework for data-layout optimizations. *SIGPLAN Not.*, 37(1):140–153, 2002.

[61] 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.

[62] Anil B. Somayaji. *Operating System Stability and Security through Process Homeostasis*. PhD thesis, University of New Mexico, July 2002.

[63] J. M. Spivey. Fast, accurate call graph profiling. *Softw. Pract. Exper.*, 34(3):249–264, 2004.

[64] B. Sprunt. Pentium 4 performance-monitoring features. *Micro, IEEE*, 22(4):72–82, Jul/Aug 2002.

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

[66] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a Java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195, 2001.

[67] hprof: an open source java profiler. `http://java.sun.com/developer/technicalArticles/Programming/HPROF.htm`

[68] xprof: Internal profiler for hotspot. `http://java.sun.com/docs/books/performance/1`

[69] Walter F. Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[70] Kapil Vaswani, Aditya V. Nori, and Trishul M. Chilimbi. Preferential path profiling: compactly numbering interesting paths. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 351–362, Nice, France, 2007. ACM.

[71] Abdul Waheed and Diane T. Rover. A structured approach to instrumentation system development and evaluation. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 30, New York, NY, USA, 1995. ACM Press.

[72] John Whaley. A portable sampling-based profiler for java virtual machines. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 78–87, New York, NY, USA, 2000. ACM.

[73] Yourkit, llc: Commercial java profiler. `http://www.yourkit.com/`.

[74] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi. Accurate, efficient, and adaptive calling context profiling. *SIGPLAN Not.*, 41(6):263–271, 2006.

[75] Craig B. Zilles. Benchmark health considered harmful. *SIGARCH Comput. Archit. News*, 29(3):4–5, 2001.