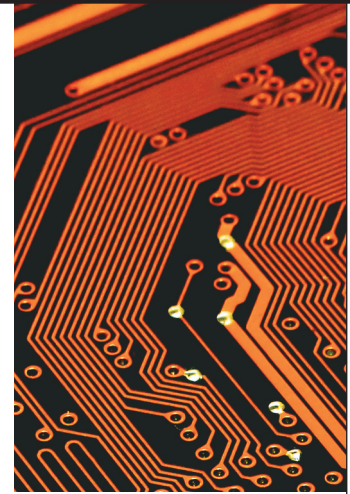


# The Effect of Omitted-Variable Bias on the Evaluation of Compiler Optimizations



- ➔ **Todd Mytkowicz and Amer Diwan**, *University of Colorado at Boulder*
- ➔ **Matthias Hauswirth**, *University of Lugano*
- ➔ **Peter F. Sweeney**, *IBM T.J. Watson Research Center*

**Experiments that do not account for all relevant variables can produce misleading results. The authors identify two variables that compiler optimization evaluations often omit, resulting in significant omitted-variable bias, and discuss how to identify new OVB sources and incorporate them in an experimental design.**

**E**xperimentation is central to science, influencing all stages of scientific investigations. At the beginning of an investigation, we conduct experiments to understand a system's nature, perhaps to identify performance bottlenecks or other weaknesses. Later on, we conduct experiments to evaluate innovations to the system or to test hypotheses.

An improperly designed experiment can produce misleading results. For example, suppose we poll residents of a small US town and, based on this poll, predict a national election's outcome. The US is home to many diverse groups with different opinions and agendas; any small town contains only some of these groups, and not in the same proportion as in the entire country. Thus, this experiment is flawed because it does not consider the sample's political bias.

If an experimental design does not adequately account for all variables that may influence its outcome (*relevant variables*), the unaccounted-for relevant variables may bias the results. In statistics, this is known as *omitted-variable bias* (OVB).<sup>1</sup>

OVB is a problem common to all experimental sciences. For example, suppose we posit  $\iota$  as a way to improve pro-

gram performance. We conduct an experiment to evaluate  $\iota$  in a single experimental setup—program, computer, input, and so on—and, based on this experiment, conclude that  $\iota$  is generally useful. This experiment is flawed in the same way as the previous example. Using only a single experimental setup ignores the influence of many relevant variables, such as the amount of machine memory. If our setup is atypical, the results will not apply to commonly used setups.

Unlike *random errors*,<sup>2</sup> we cannot eliminate OVB by running an experiment many times and summarizing the results using statistical techniques such as confidence intervals. This is because OVB introduces *systematic error*<sup>2</sup>—that is, the results are repeatable. For example, if we poll the small town again, we can expect the same misleading results.

OVB occurs in at least one area of computer science: compiler optimization evaluation. Unfortunately, it may lead to an incorrect conclusion that an optimization is effective when in reality it is not, and vice versa. In particular, two variables contribute to OVB when evaluating compiler optimizations: *environment size* and *link order*. These variables are significant—they affect the evaluation by tens of percent; commonplace—they cause OVB in a

wide variety of situations; and nonobvious—prior work does not adequately account for them.

## REPEATABLE, REPRODUCIBLE, OR USEFUL?

What do we look for in experimental results? To answer this question, we define three terms: repeatable, reproducible, and useful.

By *repeatable*, we mean that we can repeat an experiment using the same setup and yield consistent results. For example, we measure the speedup due to optimization  $\iota$  on a benchmark suite. We repeat the measurements  $N$  times to obtain speedups  $\sigma_1, \dots, \sigma_N$ . Using statistical techniques, we conclude that the optimization gives an average speedup of  $\sigma \pm \delta$  with 95 percent confidence, where  $\sigma$  is the arithmetic mean of  $\sigma_1, \dots, \sigma_N$  and  $\delta$  is the 95 percent confidence interval.

This experiment is repeatable if we pick a large enough  $N$  to ensure statistically significant results. The multiple runs along with the confidence intervals account for run-to-run variations due to, for example, operating system activity. Repeatability ensures that the results are not outliers—say, due to a fluke interaction with some OS activity.

By *reproducible*, we mean that others can repeat our experiment and obtain results consistent with ours. Reproducibility is essential for scientific advancement: if other scientists cannot reproduce our experiments, our results will have little credibility or impact.

While the above experiment is repeatable, it may not be reproducible. Specifically, others may use a different setup—for example, their machine may have more or less memory—and thus not obtain results consistent with ours. We can improve the experiment's reproducibility as follows: We evaluate the optimization as before. However, in addition to reporting the mean speedup and confidence intervals, we also report all details of our setup: the software and hardware, including the account we used and the keystrokes typed to run the experiment. Others can now reproduce the same setup, repeat our experiment, and obtain results consistent with ours.

By *useful*, we mean that “real” users of our optimization should get results consistent with our reported results. While our modified experiment is now reproducible, it may not be useful if its results apply only to our exact experimental setup and others do not use it.

## EXAMPLE OF OVB

Compilers implement many optimizations, ranging from the fast and commonly useful (optimization level O1) to the slow and more speculative (O2) to specialized optimizations (O3). To understand whether a given optimization level's benefit is worth its cost, we must evaluate the different optimization levels with respect to one another. Here we focus on comparing the benefit of O3 to that of O2 using the GNU C Compiler (GCC).

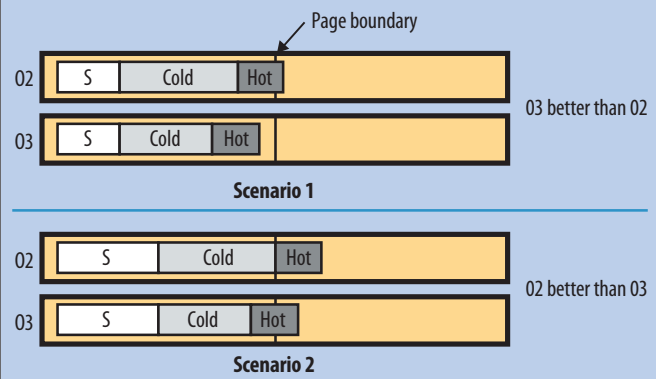


**Figure 1.** Omitted-variable bias in compiler optimization evaluation resulting from  $S$ , a setting determined by a relevant variable that affects the code's position in memory. In Scenario 1, the hot (performance-critical) code straddles a page boundary when we use compiler optimization level 2 (O2). Optimization level 3 (O3) does not affect the hot code any differently from O2; however, O3 shrinks the cold (non-performance-critical) code. Because of this, the hot code fits on a single page with O3 and thus incurs fewer page faults than with O2. Consequently, we wrongly conclude that O3 produces faster code than O2. In Scenario 2, the relevant variable's setting causes the hot code to fit on a single page with O2 but to straddle a page boundary with O3. Consequently, we wrongly conclude that O3 degrades performance compared to O2.

For such an evaluation to yield useful results, the experimental design should account for all OVB sources. If we omit relevant variables from consideration, we may incorrectly conclude that O3 is useful compared to O2 when it is not, or vice versa.

For example, compiler optimizations can either intentionally or unintentionally affect memory layout, which is critical to overall performance. Therefore, any variables that affect memory layout will interact with compiler optimizations. Not including these variables in the experimental design will cause OVB and bias the results.

Figure 1 illustrates how a variable that affects memory layout can impact the comparison of O3 with O2. Suppose we're evaluating a kernel benchmark that fits on two pages in memory. Hot is performance-critical code, Cold is non-performance-critical code, and  $S$  is a setting determined by a relevant variable that affects the code's position in memory.

In Scenario 1, the hot code straddles a page boundary when we use O2. O3 does not affect the hot code any differently from O2; however, O3 shrinks the cold code. Because of this, the hot code fits on a single page with O3 and thus incurs fewer page faults than with O2. Consequently, we conclude that O3 produces faster code than O2. In Scenario 2, the relevant variable's setting causes the hot code to fit on a single page with O2 but to straddle a page boundary with O3. Consequently, we conclude that O3 degrades performance compared to O2.

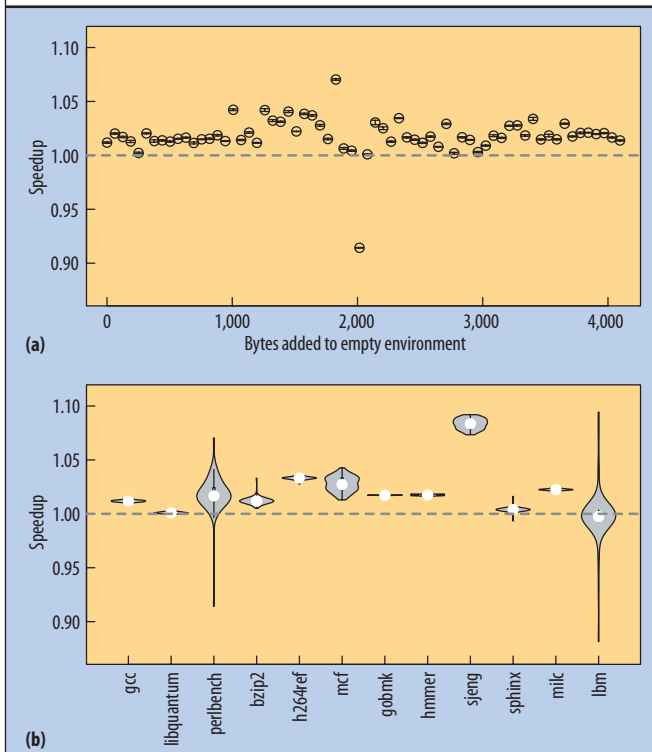


**Figure 2. Effect of Unix environment size on speedup of O3 versus O2 on a Core 2 workstation for (a) perlbench and (b) all benchmarks. Omitting environment size from our experimental design caused significant OVB for some programs, such as perlbench and lbm, and modest OVB for others.**

Both conclusions are wrong in general: because O3 has no effect on the hot code, it should have minimal positive or negative effect on performance. Moreover, both are repeatable but not reproducible or useful: the conclusions are each specific to the particular value of a relevant variable. Our experiment thus suffers from OVB by ignoring a relevant variable that determined code layout and thus led us to an incorrect conclusion.

This phenomenon is not limited to interactions with pages: there are numerous other buffers in hardware, such as caches and fetch buffers, and alignment with them can similarly affect evaluation of compiler optimizations.

### OVB IS SIGNIFICANT AND COMMONPLACE

We inspected many aspects of our experimental setup to determine which variables may affect memory layout. We identified two variables:

- *Environment size.* Many OSs provide environments that hold state—for example, the PATH environment variable—all programs can use. The environment occupies space in memory for every program whether or not a program actually accesses the environment's state. Thus, the environment size can affect

the memory layout of all code and data allocated after the environment. In particular, the environment size affects the start address of data allocated in a program's call stack.

- *Link order.* The order in which we link code determines the code layout in memory. For example, in the Unix environment “ld a.o b.o” places the code of a.o before b.o but the command “ld b.o a.o” does just the opposite.

To determine whether omitting these variables causes significant OVB, we investigated the same question shown in Figure 1: does O3 produce better code than O2 when using GCC? We quantified the “better code” using speedup—the ratio of the execution time with O2 over the execution time with O3. Speedups higher than 1 indicate that O3 produces faster code than O2; speedups less than 1 indicate that O3 produces slower code than O2.

We conducted our experiments on the 12 C benchmarks from the SPEC CPU2006 suite using Linux 2.6.25, GCC 4.1.3, and a 2.4-GHz Intel Core 2 workstation with 8 Gbytes of RAM and four cores. We let the OS schedule the single-threaded benchmarks on the cores. We repeated each experiment five times to obtain statistically significant results.

We have reproduced similar experiments in other hardware setups—for example, using a Pentium 4 workstation. Other OSs, including Windows XP, have analogies to environment variables and link order, though they may be hidden under graphical-user interfaces; thus we believe, but have not yet confirmed, that our results generalize to other OSs.

### Environment size

The environment size affects the call stack's starting address—it affects how, for example, local variables allocated on the stack align in the different hardware buffers.

Figure 2a shows the speedup for a single program, perlbench (a Perl interpreter), as we changed the Unix environment size by concatenating to the value of a dummy variable—a variable that we created and no program actually uses. A point (x,y) on this scatter plot indicates that when the environment is x bytes, the speedup is y. The leftmost point is for an empty environment; each subsequent point extends the environment by 63 bytes. The whiskers give the 95 percent confidence intervals around the mean. Because these intervals are tiny (the whiskers are not visible most of the time), the data is repeatable with minimal variation between runs.

Some environment sizes—for example, size 1,827 bytes—show significant speedup for O3, while others—for example, size 2,016 bytes—show significant slowdown for O3. Even if we ignore these two points as outliers (though both are repeatable and not random noise), we still see

a 5 percent swing in speedup as a result of changing the environment size. In other words, omitting the environment size caused significant OVB; its effects were large enough to obfuscate our results. Worse, there is no obvious trend: increasing the environment size either increased or decreased speedup, suggesting that this OVB was unpredictable.

Figure 2b uses violin plots to summarize the results for all C programs in the SPEC CPU2006 benchmark suite. Each violin summarizes data for the different environment sizes for one benchmark—for example, the perlbench violin summarizes Figure 2a. The white dot in each violin gives the median, and the thick line through the white dot gives the interquartile range. The width of a violin at  $y$  on the  $y$ -axis is proportional to the number of times we observed  $y$ . The height of a violin gives the range of speedups we obtained by changing the environment size.

From the height of the violins, we see that omitting environment size from our experimental design caused significant OVB for some programs, such as lbm and perlbench, and modest OVB for others. The most extreme change occurred with lbm, whose speedup ranged from 0.88 (a significant slowdown due to O3 optimizations) to 1.09 (a healthy O3 speedup). Thus, one environment size led us to conclude that O3 significantly improves performance over O2, but another environment size led us to conclude the exact opposite. Moreover, we found that environment sizes interacted differently with different programs—for example, the environment size that gave the best speedup of O3 over O2 for one program did not give the best speedup for another program.

The violins for libquantum, perlbench, sphinx, and lbm straddle 1.0. Thus, for these four benchmarks, we can conclude that O3 may or may not be useful—that is, different environment sizes may lead to contradictory results.

In summary, something as seemingly innocuous as altering the environment size changed our conclusions by tens of percent; this is large enough to obfuscate many experimental results.

## Link order

The order in which we provide binary files to the linker affects code layout. We often do not give much thought to the link order—for example, we may use `ld *.o` to get the alphabetical order, or whatever the makefile happens to provide—and thus frequently omit it from our experimental design.

Figure 3 illustrates the effect of link order on the speedup of O3 versus O2. To obtain this data, we compiled each benchmark 33 times; the first time we used the link order specified by the makefile, the second time we used an alphabetical link order (the .o files appeared in alphabetical order), and the remaining times

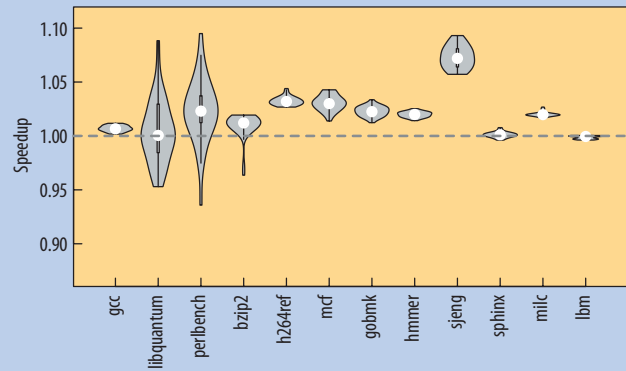


**Figure 3. Effect of link order on speedup of O3 versus O2 on a Core 2 workstation for all benchmarks. Omitting link order from our experimental design caused significant OVB for some programs, such as libquantum and perlbench, and modest OVB for others.**

we used a randomly generated link order. Each violin summarizes data for the different link orders for one benchmark.

From the height of the violins, we see that omitting link order from our experimental design caused significant OVB for some programs, such as libquantum and perlbench, and modest OVB for others. The most significant change in speedup occurred with perlbench, whose speedup ranged from 0.93 (a significant slowdown) to 1.09 (a significant speedup). Thus, one link order led us to conclude that O3 significantly improves performance over O2, but another link order led us to conclude the exact opposite. Moreover, we found that link order interacted differently with different programs—for example, the link order that gave the best speedup of O3 over O2 for one program did not give the best speedup for another program.

The violins for libquantum, perlbench, bzip2, sphinx, and lbm straddle 1.0. Thus, for these five benchmarks, we can conclude that O3 may or may not be useful—that is, different link orders may lead to contradictory results.

As with environment size, the link order of binary files changed our conclusions by tens of percent, which again is large enough to obfuscate many experimental results.

## Other OVB sources

While environment size and link order are the dominant OVB sources in our study, others are also worth exploring. For example, the malloc/free implementation or the size of available memory in a garbage-collected system<sup>3</sup> can also cause OVB. Experience in other sciences suggests that even with the best efforts there will still be some residual sources of OVB.<sup>4,5</sup> Thus, it's critical for the computer science community to continue identifying new and significant sources.

## WHY WORRY ABOUT OVB?

Can OVB be ignored? Unfortunately not: we found OVB to be

- *significant*—OVB affected our data by tens of percent;
- *commonplace*—we observed OVB on multiple microprocessors (the IBM Power4, the Intel Pentium 4 and Core 2, and a simulated Alpha microprocessor using the m5 simulator<sup>6</sup>), on another compiler (icc, Intel's C compiler), and on most of the 12 SPEC CPU2006 C benchmarks; and
- *unpredictable*—we saw no obvious trends with OVB; for example, the environment size that gave the best speedup for O3 on one microprocessor and benchmark combination usually did not yield the best speedup for another benchmark or microprocessor.

Does this scenario sound familiar? You run a set of experiments and submit them to a conference. Three months later, the conference accepts your paper but asks for more data. When you try to reproduce your experiments, you get different results. In such cases, it's tempting to blame the discrepancy on some random event, such as a daemon or network activity, but such phenomena are easy to ac-

**Omitted-variable bias is significant, commonplace, and unpredictable.**

count for using statistical methods such as confidence intervals. The discrepancy between old and new data may have more to do with OVB than with random errors.

If you cannot reproduce your own experiments, do not expect others to be able to reproduce them either. They may have different environment variables—for example, the HOME variable, which records a user's home directory, will almost certainly be different—or use a different link order. Both of these factors can easily affect experimental outcomes.

In short, OVB is not something that affects only strange “corner” cases. You've likely already experienced it, perhaps without realizing it.

## INEFFECTIVE APPROACHES FOR DEALING WITH OVB

A recent survey<sup>7</sup> shows that computer scientists attempt to work around OVB by holding constant all relevant variables or by using multiple workloads. Unfortunately, neither of these techniques is generally effective.

### Hold constant all sources of OVB

If we can identify all relevant variables, we can simply hold them constant and report them with our results; such

variables are called *control variables*. For example, we may pick a particular environment size and link order, use it for our experiments, and report it with our results. As indicated earlier, this approach produces reproducible but not necessarily useful results.

### Use many workloads

Instead of evaluating a compiler optimization using a single benchmark or suite, we can use many benchmarks or suites and report averages over all the benchmarks. The OVB may bias results for some benchmarks in one way and other benchmarks in another way; thus, hopefully, the averaging will cancel out the effects of OVB.

To see if this was the case, we computed the average speedup of O3 across all 12 of the SPEC CPU2006 C benchmarks using many different experimental setups that varied in environment size and link order. We found that the average speedup ranged from 1.01 (a small speedup) to 1.07 (a significant speedup) as we changed the setup.

Thus, while a sufficiently large and diverse set of benchmarks can theoretically cancel out the effects of OVB, it was not true for the SPEC CPU2006 C benchmarks.

## EFFECTIVE COUNTERMEASURES

To effectively deal with OVB, we identify the relevant variables omitted from our experiments and incorporate these variables in our experimental design.

### Identifying omitted variables

Our approach addresses only omitted variables that significantly impact experimental outcomes. For example, we demonstrated that environment size and link order affect performance by tens of percent, which is large in the context of many compiler optimizations. In addition, we focus only on omitted variables that are not “constants” in practice. For example, because most embedded platforms use one factory-installed memory size, omitting memory size will not significantly degrade the generality of experimental results.

Toward this end, we conduct user trials to discover what omitted variables vary between users. Specifically, we have many individuals carry out the same experiment with the same setup, and from each experiment we collect both the results and setup details: microprocessor, motherboard, OS version, and so on. If our design does not suffer from OVB, then all experiments should yield the same outcome; if they do not, then we look at the underlying system details and try to determine the relevant variables that we've omitted. Once we have found an omitted variable, we incorporate it into our design and repeat the experiments.

On a small scale, this is how we determined that omitting environment size and link order from our experimental design caused OVB. There are, of course,

challenges to generalizing this approach; these challenges—for example, identifying a representative subset of the population—are common to all user trials. The user trials also discover all relevant omitted variables, but it will identify those that are different between experimental setups.

### Incorporating omitted variables

Once we have identified a relevant omitted variable, we incorporate it in our experimental design—thus it is no longer “omitted” and actually becomes an independent variable. The basic approach we use is random sampling; specifically, we use many different values for the relevant variables. For example, we may generate setups that differ in link order and environment size, then run our experiment on each setup and record the data.

The outcome of this step is not a single data point but a distribution of data points—that is, one point for each experimental setup. We can then use standard statistical techniques to draw conclusions from the distributions. For example, using this technique we determined that with a 95 percent confidence interval, O3 provides a modest ( $1.007 \pm 0.003$ ) speedup over O2 for the perlbench program on a 2.4-GHz Core 2 workstation with Linux 2.6.25, GCC 4.1.3, and 8 Gbytes of memory. OVB does not impact this data, at least with respect to environment size and link order.

**P**roperly designed experiments that produce valid data and ensure valid conclusions are essential for scientific progress. For this reason, scientists expend considerable effort in experimental design. One aspect of careful design is avoiding omitted-variable bias. Results from experiments with OVB are not necessarily reproducible and thus are not generally useful.

OVB probably affects most experimental work in computer science, and we have identified two relevant variables that evaluations of compiler optimizations commonly omit. Fortunately, techniques adopted from other sciences can prevent OVB. It is critical for us, as a community, to identify new and significant sources of OVB to improve the science in computer science. 

### Acknowledgments

We thank Evelyn Duesterwald, Steve Fink, Adam Ghobarah, Mike Hind, and Dick Sites for valuable discussions and feedback on this article, and Mary Jane Irwin for encouraging us to write it. This work was supported by Nation Science Foundation grant NSF CSE-0509521 and DARPA grant number HR0011-07-9-0002. Any expressed opinions, findings, and conclusions or recommenda-

tions are ours and do not necessarily reflect those of the sponsors.

### References

1. W.H. Greene, *Econometric Analysis*, 6th ed., Prentice-Hall, 2007.
2. J.R. Taylor, *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements*, 2nd ed., University Science Books, 1997.
3. S.M. Blackburn, P. Cheng, and K.S. McKinley, “Myths and Realities: The Performance Impact of Garbage Collection,” *ACM SIGMETRICS Performance Evaluation Rev.*, June 2004, pp. 25-36.
4. R.P. Feynman and R. Leighton, “Surely You’re Joking, Mr. Feynmann!” *Adventures of a Curious Character*, W.W. Norton & Co., 1997.
5. J.P.A. Ioannidis, “Contradicted and Initially Stronger Effects in Highly Cited Clinical Research,” *JAMA*, 13 July 2005, pp. 218-228.
6. N.L. Binkert et al., “The M5 Simulator: Modeling Networked Systems,” *IEEE Micro*, July/Aug. 2006, pp. 52-60.
7. T. Mytkowicz et al., “Producing Wrong Data without Doing Anything Obviously Wrong!” *Proc. 14th Int’l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS 09)*, ACM Press, 2009, pp. 265-276.

*Todd Mytkowicz received a PhD in computer science from the University of Colorado at Boulder and will be joining the Research in Software Engineering (RiSE) group at Microsoft Research in fall 2010. His research interests include performance analysis and measurement, experimental design, and optimization. Contact him at [todd.mytkowicz@colorado.edu](mailto:todd.mytkowicz@colorado.edu).*

*Amer Diwan is an associate professor in the Department of Computer Science at the University of Colorado at Boulder and a member of the technical staff at Google. His research interests include performance analysis of large-scale computer systems and programmer productivity tools. Diwan received a PhD in computer science from the University of Massachusetts Amherst. He is a member of the ACM. Contact him at [diwan@cs.colorado.edu](mailto:diwan@cs.colorado.edu).*

*Matthias Hauswirth is an assistant professor in the Faculty of Informatics at the University of Lugano, Switzerland. His research focuses on performance measurement, understanding, and optimization. Hauswirth received a PhD in computer science from the University of Colorado at Boulder. He is a member of IEEE, the IEEE Computer Society, and the ACM. Contact him at [matthias.hauswirth@unisi.ch](mailto:matthias.hauswirth@unisi.ch).*

*Peter F. Sweeney is a research staff member in the Programming Technologies Department at IBM T.J. Watson Research Center in Hawthorne, New York. His interests include object-oriented optimization, dynamic compilation, and performance analysis. Sweeney received an MS in computer science from Columbia University. He is a senior member of the ACM and a member of IEEE. Contact him at [pfs@us.ibm.com](mailto:pfs@us.ibm.com).*



Selected CS articles and columns are available for free at <http://ComputingNow.computer.org>.