Languages and Performance Engineering: Method, Instrumentation, and Pedagogy

Doug Lea
SUNY Oswego
dl@cs.oswego.edu

David F. Bacon
IBM Research
dfb@watson.ibm.com

David Grove
IBM Research
groved@us.ibm.com

Abstract

Programs encounter increasingly complex and fragile mappings to computing platforms, resulting in performance characteristics that are often mysterious to students, practitioners, and even researchers. We discuss some steps toward an experimental methodology that demands and provides a deep understanding of complete systems, the necessary instrumentation and tools to support such a methodology, and a curriculum that teaches the methodology and tools as a fundamental part of the discipline.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Programming languages curriculum

1. Introduction

A developer has a task to achieve, and uses a programming language, along with available library components and associated program development support tools, to express a solution. The language, its libraries, and tools provide a conceptual basis for a solution. When a correct solution is readily arrived at and performs as expected, all is well.

Different programming languages present different models of computation, that spur developers to solve problems using the idioms, coding techniques, and design patterns they have learned for structuring program components – traditionally: functions, data plus procedures, (passive) objects (plus threads), communicating processes, and so on. Often enough, none of these simple models entirely fit the problem at hand. And even more often, the resulting programs bear fragile relationships to the platforms they execute on.

The performance of two similar programs under varying mappings might differ by many orders of magnitude depending on issues such as branch prediction, cache misses, cache contention, the placement of memory fences, mappings of threads to cores or processors, OS scheduling policies, memory footprint, resolution of virtual dispatches, loop parallelization, SIMD or GPU parallelism, inlining, boxing of primitives, garbage collection policies, interpretation vs machine code compilation, reflection, opportunistic synchronization elimination, contention retry policies, remote messaging, load balancing, memory, file, and network layout, and so on.

Mapping programs onto systems with increasingly complex performance characteristics forms a problem of our own creation. More than any other field, computer science is a *science of the artificial* [5]. Natural science is primarily concerned with the study of existing systems. In contrast, computer science deals with both the construction and the subsequent behavioral study of a limitless set of possible designs and implementations for the solution of a given problem.

2008 SIGPLAN Workshop on Programming Language Curriculum , May 29–30, 2008, Cambridge, MA, USA.

As systems grow in complexity, performance engineering becomes a more central area of study and instruction in its own right. At the same time, a basic understanding of performance is becoming essential for all computer scientists and software engineers. Lack of sufficient awareness of the phenomena listed above (among others) limits a student's ability to usefully employ the algorithmic complexity tools (Big-Oh etc) taught in the CS curriculum. Without performance models describing the mapping from languages to machines, or a systematic methodology for measuring performance, the question of whether a program runs fast or slow does not have a defensible answer; complexity analysis is at best a rough guide. As a result, few students are able to intentionally write efficient programs, and many students unintentionally write incurably slow ones without understanding or even noticing the consequences. This does nothing to improve the current situation in practical software development of many projects being delayed or scrapped due to performance problems, and the elevation to "star" status of those few who can predictably engineer software with predictably good performance. Even the best researchers often have only a superficial understanding of the systems they have built. It is all too common to read published papers with uninformative passages such as: "our optimization produced a mean speedup of 8.3%; two benchmarks slowed down considerably, which we suspect is due to cache effects".

In response, the language implementation research community (and, more broadly, "systems" fields of computing) must undertake fundamental changes in research, practice, and education, to establish:

- A methodology that demands and provides a deep understanding of complete systems.
- The necessary instrumentation and tools to support such a methodology.
- A curriculum that teaches the methodology and tools as a fundamental part of the discipline.

2. Method

The essence of building systems is abstraction and hierarchical organization. In nature, we have chemicals, atoms, and (sub-atomic) particles. Working at one level of abstraction, one can largely treat the underlying layers as obeying fairly simple and predictable rules. In computer science, we have machine architecture, operating system, language run-time, network, and application. These components and the abstractions of the underlying systems that they present have proven enormously successful for the creation of large, rich applications. However, breakdowns in the isolation of these abstractions at the implementation level have had a continuing impact on computing research and practice.

Thirty years ago, the abstractions governing the mappings of programs to systems did a pretty good job of insulating the upper layers of the hierarchy from their implementation. For instance, most machines had only one processor, and transistors were scarce enough that CPUs were relatively simple, so the time it took to execute a single instruction was highly predictable. A programmer could look at a page of assembly code and accurately predict how long it would take to execute. Reliance on such simple leak-proof abstractions also characterized the *descriptive* era of computer science research methodology. From roughly 1960 to 1975, papers usually described either an algorithm with a mathematical description of its performance, or a system implementation technique with a qualitative description of its benefits.

Within some areas of computing, especially real-time systems, there have been compelling arguments (notably by Lee [3]) that implementors must continue to provide dependable abstraction barriers that support the reliable construction of applications. However, work in most areas of computing has long aimed to escape the confines of the "real time means real slow" trap. As a result, once the most obvious uniform performance enhancement techniques were exhausted, more narrow, special-case refinements proliferated, and the limitations of mathematical modeling became apparent. This in turn led to a more empirical approach to computer science research. In the *reductionist* era, roughly 1975 to 1990, experimental papers turned to the measurement of running systems, but tended to focus on the performance of small components; for example, PLDI papers evaluated optimizations on a few "kernels" such as the LINPACK benchmarks, and manufacturers quoted performance in MIPS.

The reductionist approach became less tenable as even the designers of chips became increasingly unable to account for CPU throughput. Features such as hardware multi-threading, speculative execution, and non-uniform memory hierarchies succeeded in driving up average-case performance, but at the expense of uniform predictability. Implementors became decreasingly able to ensure that program components are *always* fast, so they settled for *usually* fast, widening the gulf between typical and worst-case performance. Thus, while functional semantics remained compositional, performance did not. Anomalies commonly seen in practice reflect the failure to hierarchically isolate the mechanics underlying these abstractions. Around 1990, such observations led to a shift to a *holistic* methodology in computer science research, where the effects of implementation techniques were evaluated by their effects on a set of (putatively) realistic benchmarks. In programming languages, this was manifested by the preponderance of PLDI papers with bar graphs showing end-to-end speedups for 8 or so benchmarks. This trend was paralleled in industry by the rise of the SPEC benchmarks, which became widely used for research evaluation as well.

The products of holistic methodology now bring us to an era in which there is often an inverse relationship between the quality and predictability of performance. As the average performance of systems improves, the predictability of performance (i.e., the quality of its performance model) worsens. The price users pay for usually-good performance is a complex and fragile performance model that varies over time as new implementation techniques are discovered for both hardware and software. Performance has become *chaotic* in the technical sense of extreme sensitivity to initial conditions: The further a program is away from the targeted design center of a system, the more the programmer must be aware of the possibilities of steep drop-offs, and must learn to "program between the lines" [2] to cope with *idiot savant* compilers, runtime systems, and hardware employing bags of tricks that work well when they apply, but do not always apply. These issues will move further to the forefront as platforms become even more diverse, executing programs using combinations of multicores, multiprocessors, accelerators, FPGAs, clusters etc; each of which provide opportunities for different kinds of narrowly targeted optimizations.

Current levels of complexity and porousness of abstraction barriers have reduced the value of a purely holistic methodology. For a representative example, it has been observed on a number of systems that execution time effects due to using `-O3` versus `-O2` compiler optimization levels in a C program can vary by up to +20% to -11% merely by changing the length of shell environment variables [4]. The majority of compiler optimizations presented in the literature achieve speedups of a similar or smaller magnitude. Our confidence that such new techniques are even useful declines accordingly. The net effect is akin to that of a chemist witnessing quantum-mechanical phenomena breaking the "abstraction barrier" of atoms, causing molecules to fade out of existence for a while. If this were a common phenomenon, progress in chemistry would nearly halt.

As performance variability increases, there is a consequent need for language, component, and system designers and implementors to provide means of reducing this variability; often by providing lower-level constructs and APIs that allow expert programmers to better control mappings onto platforms. Language and their implementations will increasingly need to include meta facilities for automating (perhaps platform-specific) program transformations or tunings with predictable performance outcomes.

At the same time, there is a need for methodologies closer to those of biology than of chemistry, focused more on *explanation* than prediction. Why does a simple change to a program, or even a compiler switch cause a dramatic change in performance? Why does a component run slower as the program using it runs longer? Answers to these and hundreds of similar questions require advances in metrics and support tools.

3. Instrumentation

Isaac Newton said "What can be measured can be understood". If we aspire to make computer science truly be a science, and produce software engineers who truly perform engineering, we have a responsibility to create the instruments with which to measure computer systems. Researchers must develop a comprehensive methodology for instrumenting and measuring systems, and for properly performing statistical analysis on

the results of those measurements. Many of the ingredients are common to just about all empirical science. However, a prerequisite that is often lacking in computer systems is the ability to instrument, measure and observe systems in the first place.

The more quickly and accurately one can measure and evaluate a system, the more completely one can study and improve it. This is obvious; yet the way in which this manifests in practice can be quite surprising in terms of quality of experiment and productivity in development. While as rationalists we tend to think of experimental factors in absolute terms (a factor either needs to be measured or it doesn't) the truth of the matter is that the ease with which a system can be introspected has an enormous effect on our productivity as both developers and as scientists. To capture this we introduce a new bogometric: *hypotheses per minute* (HPM), the number of hypotheses about a computer system that can be evaluated per minute. While tongue-in-cheek, it will be familiar to anyone who has ever tried to understand the surprising performance of a program. Many diverse factors can affect HPM: the quality of the instrumentation, the availability of instrumentation for a particular subsystem, the ability to correlate information from different sources, the availability of different visualizations of the data, and whether it is possible to view (and change the view) of the data interactively. HPM provides a way for us to try to decide what to implement next from this diverse and otherwise incomparable set of possible features.

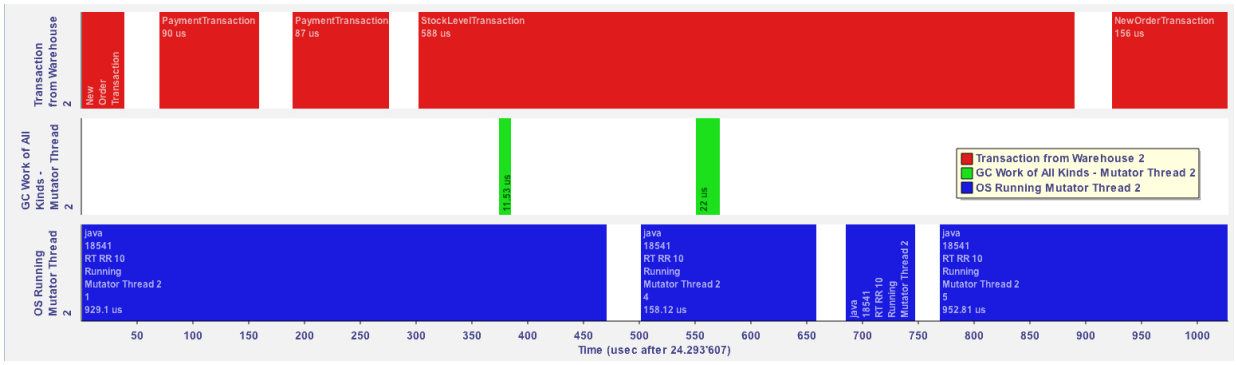
We illustrate by way of some experiences of two of us (Bacon and Grove) developing and using the TuningFork [1, 6] tool for the collection, visualization, and correlation of trace data from multiple levels of a computer system. The examples below describe our use of the tool in the development of a real-time multi-processor garbage collector in a Java virtual machine running on a NUMA multiprocessor. Our goal here is not to advocate for TuningFork in particular, but rather to illustrate the importance of such tools and their use as a scientific instrument for the study of total system behavior caused by complex interactions across multiple sub-systems.

Sometimes the requirements for instrumentation are obvious. For instance, when developing a real-time garbage collector, it is clearly important to understand the kernel's scheduling decisions. Unfortunately, developing such instrumentation in a way that is accurate, inexpensive, and precise (despite for instance unsynchronized cycle timers on an SMP), is labor intensive. Developers and researchers commonly face the question of whether it is worthwhile to spend the required effort (several person-months) to add this capability if it is not already provided. Increasingly, the answer is yes. We were completely taken aback by how a simple change to TuningFork's user interface – allowing data streams to be dragged and dropped into an existing figure – caused an enormous increase in productivity. We found that as users we simply tended to explore more hypotheses because the user interface had made it easier to do. This feature only took a few hours to add, but took us months to stumble upon.

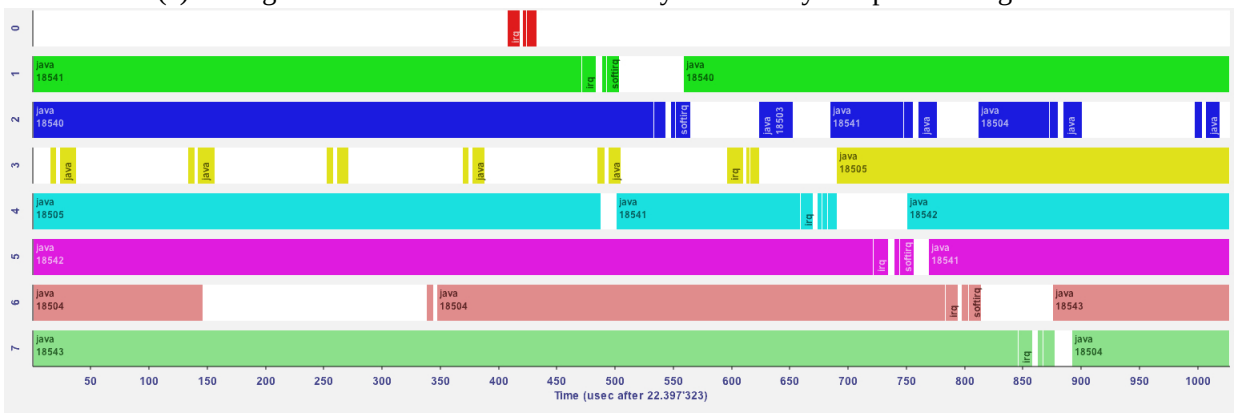
A further requirement for tools is *verticality*, that is, the ability to collect and evaluate measurements across the different sub-systems that make up the execution environment as a whole. As an example of how we apply this in practice, consider Figure 1, which shows how we diagnosed an unusually long SPECjbb transaction running on top of a real-time JVM running on top of a RHEL5 RT real-time Linux kernel running on an 8-way NUMA LS-41 blade with 4 dual-core AMD CPUs.

In Figure 1(a) we begin by looking at the transaction data for Warehouse 2 (top row) from our application-level trace, which shows a long StockLevelTransaction (588μs, as opposed to a norm of 350μs). Our first hypothesis was that this was caused by some sort of interference by the garbage collector, so we included data from the JVM trace showing GC activity on that thread (second row). We see that there is indeed some GC activity, but it is extremely short (under 34μs) and does not account for the magnitude of the problem. We then looked for activity by the JIT, the class loader, and the page fault handler – all of which were negative (for brevity, they are not shown in the figure).

Our next hypothesis was that the thread is being preempted by the OS (even though it is running at high priority), so we look at data which cross-correlates the Linux kernel trace with the JVM trace to show exactly when the JVM's thread was being run by the operating system (third row). This shows that the user's



(a) A long StockLevelTransaction caused by GC activity and process migration



(b) Migration was caused by “rolling” OS timer interrupts and multiple priority-based preemptions

Figure 1. GC activity and OS scheduling for a long transaction

thread was preempted three times during the transaction. In fact, we also see that the thread was migrated from CPU 1 to 4 to 2 to 5. Why did the OS schedule the thread so oddly? The answer can be determined from Figure 1(b), which shows all activity on each of the 8 CPUs. From this, we can see that the thread that runs Warehouse 2 (Java Mutator Thread 2 with OS thread id 18541), migrated in response to interrupt servicing (irq and softirq – these are the 1ms OS scheduler timer interrupts). Interrupts run at priority 50, so they preempt the Java thread running at real-time priority 10. However, because the IRQs do not occur simultaneously, but instead in a “waterfall” about 50μs apart starting on processor 0, the Java thread is repeatedly preempted as it migrates to a CPU which gets hit by an IRQ some tens of microseconds later.

The instrumentation, visualization, and vertical cross-correlation provided by TuningFork enabled us to diagnose this problem in under half an hour. By comparison, the “rolling irq’s” had caused a problem which a team of 6 Linux and JVM experts worked on for several weeks before finally diagnosing it with our help. For the time being though, our explanation remains incomplete, and a fuller account awaits the existence of additional tools. The observed migrations preempt the thread three times for a total of 85μs plus 34μs for GC work, leaving roughly 100μs unaccounted for. We suspect that this is due to cache effects (sic), but the group building the hardware performance counter component has not yet finished.

4. Pedagogy

A Computer Science and/or Software Engineering curriculum should teach students not only how to reason about the semantics of programs written in various languages or styles, but also how to predict, control, and/or explain their performance. This is challenging even for simple C programs. As we have shown, it becomes both more difficult and more central for languages that increase the ”semantic gap” between code and its implementation by virtue of reliance on managed runtime systems, more ”declarative” versus

procedural constructs, and dynamic type systems. While these often offer better average performance than C code, programs are also more likely to encounter more naively unexpected performance anomalies. In particular, the two step mappings in Virtual Machine based languages, first from programs to bytecodes, and then to platform-specific instructions and support code, adds a second level of potential surprise.

One impediment to teaching and learning about performance is the existence of courses that continue to isolate language implementation topics. In light of abstraction leakage and complexity, such practices may convey harmful misconceptions about how systems work. Moreover, the bulk of current undergraduate (and even graduate) education is concerned with the construction of systems in the small: systems for which specifications are concise and precise, clear optimality criteria exist, and in the absence of optimal solutions it is possible to clearly delineate the set of acceptable solutions. Unfortunately, the bulk of useful computer software consists of systems with diametrically opposite properties: they are large, and the specifications are complex or in many cases even impossible to elucidate.

Some of the ingredients of performance engineering can be profitably taught in a conventional manner – for example memory hierarchies in a Computer Architecture course, locality and scheduling in Operating Systems, and so on. While useful, the isolated study of for example caching does not itself prepare students to take locality into account when writing programs – this requires more integrative learning experiences than are common in current CS and SE courses. All students should be taught the basic performance challenges encountered by language implementors, common algorithms and heuristic techniques that cause program components to at least sometimes be fast, their typical boundary conditions, and the consequences for programmers using these languages. It is not hard to arrange a good sampling of such experiences by way of assigned programming projects.

Extended coverage of these topics might form the basis for new undergraduate or graduate courses that refactor entrenched course content boundaries to provide a more coherent picture of language support infrastructure and its implications for software development. While topics in programming languages, compilers, operating systems, and computer architecture provide some understanding of individual components, course modules are also needed to convey how they fit together and interact, sometimes on a microsecond-by-microsecond basis.

The use of performance analysis tools and instrumentation will be critical for success in educating students to investigate and quantify the complex interactions across the many layers of the computer systems with which they work. A debugger is a much better vehicle than a quiz for removing a student's misconceptions about program semantics. So too will be instrumentation-based tools for removing misconceptions about the mappings from programs to systems and their performance impact. Routine use of such tools will train students (and ourselves) to demand answers rather than to accept ignorance.

Acknowledgments

Thanks to Josh Bloch, Jim Early, and Vijay Saraswat for reading drafts and suggesting improvements. Joshua Auerbach, Perry Cheng, Matthias Hauswirth, Daniel Frampton, and David Ungar made major contributions to the development of the TuningFork instrumentation and visualization tools.

References

[1] David F. Bacon, Perry Cheng, and David Grove. TuningFork: a platform for visualization and analysis of complex real-time systems. In *OOPSLA 2007 Conference Companion*, pages 854–855, Montreal, Quebec, 2007.

[2] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, 1996.

[3] Edward A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.

[4] Todd Mytkowicz, Peter F. Sweeney, Matthias Hauswirth, and Amer Diwan. Observer effect and measurement bias in performance measurement. Submitted for publication, 2008.

[5] Hebert A. Simon. *The Sciences of the Artificial*. MIT Press, third edition, 1996. Originally published in 1969.

[6] TuningFork Visualization Platform. http://tuningforkvp.sourceforge.net.