

Vertical Profiling: Understanding the Behavior of Object-Oriented Applications

Matthias Hauswirth
University of Colorado at Boulder
Matthias.Hauswirth@colorado.edu

Amer Diwan
University of Colorado at Boulder
diwan@colorado.edu

Peter F. Sweeney
IBM Thomas J. Watson Research Center
pfs@us.ibm.com

Michael Hind
IBM Thomas J. Watson Research Center
hindm@us.ibm.com

ABSTRACT

Object-oriented programming languages provide a rich set of features that provide significant software engineering benefits. The increased productivity provided by these features comes at a justifiable cost in a more sophisticated runtime system whose responsibility is to implement these features efficiently. However, the virtualization introduced by this sophistication provides a significant challenge to understanding complete system performance, not found in traditionally compiled languages, such as C or C++. Thus, understanding system performance of such a system requires profiling that spans all levels of the execution stack, such as the hardware, operating system, virtual machine, and application.

In this work, we suggest an approach, called *vertical profiling*, that enables this level of understanding. We illustrate the efficacy of this approach by providing deep understandings of performance problems of Java applications run on a VM with vertical profiling support. By incorporating vertical profiling into a programming environment, the programmer will be able to understand how their program interacts with the underlying abstraction levels, such as application server, VM, operating system, and hardware.

Categories and Subject Descriptors

B.8.2 [Performance and Reliability]: Performance Analysis and Design Aids; C.4 [Computer Systems Organization]: Performance of Systems—*measurement techniques, performance attributes*

General Terms

measurement, performance, experimentation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Copyright ACM, 2004. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version will appear in the ACM SIGPLAN 18th Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'04), Oct. 24-28, 2004.

Keywords

vertical profiling, whole-system analysis, perturbation, hardware performance monitors, software performance monitors

1. INTRODUCTION

Compared to imperative languages, such as C/C++, modern object-oriented programming languages, such as Java, offer the benefit of increased runtime flexibility (e.g., reflection, automatic memory management), improved security properties (null pointer and array bounds checks, security policies), and a portable deployment representation. However, these features, and several others, require a sophisticated runtime system that introduces an additional layer of virtualization, such as a JVM, between the application and the operating system. Higher level programming models, such as J2EE, require a further level of virtualization in the form of an application server. Thus, in today's commercial system the application can be separated from the native hardware by several layers of virtualization: an operating system, a virtual machine, and an application server.

Although this virtualization provides several software engineering advantages, the support of this virtualization introduces obstacles to understanding application performance. For example, popular implementation techniques, such as dynamic recompilation and garbage collection, influence application behavior in a way that makes correlation of hardware performance to source code challenging. With dynamic recompilation, the same source code statement can be translated to different machine instructions at different memory locations throughout the execution of the program. Likewise, garbage collection can both relocate objects and reuse addresses, resulting in a dynamic mapping between objects and addresses.

The above issues indicate that there is a need to gather more complete profiles, containing information about system behavior on various levels (see Figure 1). We call this approach *vertical profiling*. The main goal of vertical profiling is to further the understanding of system behavior through correlation of profile information from different levels.

This paper introduces the concept of vertical profiling and demonstrates its use for understanding performance phenomena. We demonstrate that it is necessary to use vertical profiling to understand existing performance problems, and further demonstrate that vertical profiling can indeed ex-

Application
Framework
Java Libraries
Virtual Machine
Native Libraries
Operating System
Hardware

Figure 1: Layers of Virtualization

plain these problems. Through insight gained from five cases studies, we demonstrate that achieving the goal of vertical profiling requires more than capturing the mapping between virtualized and real entities.

The main contributions of this paper are

- a novel approach, vertical profiling, for capturing and correlating performance problems across multiple execution layers;
- five case studies that demonstrate how vertical profiling can be used to understand performance phenomena; and
- insight into issues that can impact the success of this performance understanding.

By incorporating vertical profiling into a programming environment the programmer will be able to understand how their program interacts with the underlying abstraction levels, such as application server, VM, operating system, and hardware.

The structure of this paper is as follows. Section 2 describes our infrastructure. Section 3 describes the methodology we use for our case studies and evaluates the overhead caused by our vertical profiler. Section 4 presents five detailed case studies that demonstrate how the infrastructure can be used to better understand program performance. Section 5 presents lessons we learned while using vertical profiling in our case studies. Section 6 covers work related to vertical profiling, and Section 7 concludes.

2. INFRASTRUCTURE

This section describes our infrastructure. Section 2.1 introduces the fundamental concepts of our vertical profiling approach. Section 2.2 describes the software performance monitors we introduced to capture the behavior of the software layers. Section 2.3 presents the implementation of the sampling infrastructure that gathers traces of performance monitor values. Section 2.4 describes our approach for analyzing those traces.

2.1 Events, States, Monitors, and Counters

The fundamental components of our vertical profiler are performance monitors. They observe (*monitor*) events and states of the system on various levels.

2.1.1 Events and States

A computer system can be viewed as a large state machine, where a state captures the values in all of memory (including registers, caches, main memory, and persistent storage). An event is an atomic occurrence in time that does not have any duration. An event usually causes the system to change to a different state. For example, an object allocation event means that the system now has less free memory, but an additional new object. Or an instruction completed event means that the processor pipeline now has one less instruction in it.

Some events can have attributes. For example, an object allocation event can have attributes specifying the object size, address, type, etc. Conversely, a cycle event (a clock tick in the CPU) has no attributes.

2.1.2 Monitors and Counters

A performance monitor observes the behavior of a system. We use the hardware performance monitors available in the processor, and we add software performance monitors to the native libraries, the virtual machine, and the Java libraries.

A performance *monitor* is a scalar variable with a value that changes over time. A performance *counter* is a special kind of performance monitor. The value of a performance counter reflects a count, usually a count of events.

2.2 Software Performance Monitors

In previous work [33] we used hardware performance monitors to analyze the behavior of applications. We found that hardware performance monitors are not enough for a complete understanding of certain performance phenomena. We concluded that we also needed information from higher layers (Figure 1) of the system. This paper thus introduces vertical profiling, adding software performance monitors (SPMs) to observe the behaviour in the layers above the hardware. Our software performance monitors cover the following parts of the system:

Application. We provide a mechanism for profiling application behavior by adding software performance monitors to applications.

Virtual Machine. We monitor several subsystems of the virtual machine:

Memory Manager. Our infrastructure implements monitors to profile allocations of arrays and objects, and growing and shrinking of the virtual machine’s heap.

Runtime Compilers. We provide software performance monitors for dynamic compilations that count the number of methods compiled, and the number of byte code and machine code instructions generated by the different compilers, and captures the compiled method ID and optimization level of the last method that is recompiled.

Synchronization. We profile synchronization operations such as locks and unlocks, notify and waits, the number of thread yields because of the need to wait for a lock, and the number of attempted and succeeded test-and-set operations.

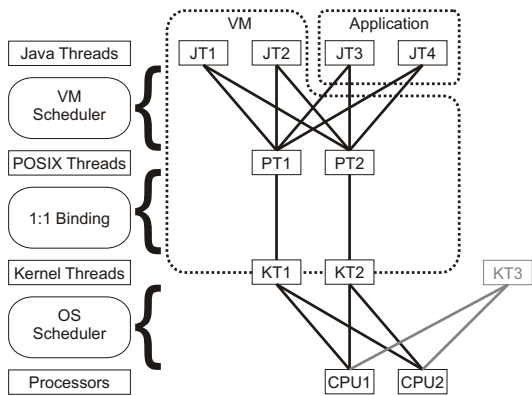


Figure 2: Scheduling in Jikes RVM

Operating System. We observe the interactions between virtual machine and operating system. Those interactions go both ways, system calls into the OS, and signals sent to the virtual machine. We provide monitors that observe virtual memory management requests, and signals caused by segmentation violations and arithmetic errors.

2.3 Implementation

Our vertical profiling infrastructure is based on the hardware performance monitor capability that exists in Jikes RVM [33]. This section summarizes the existing infrastructure and describes our extensions.

Jikes RVM [19] is an open source research virtual machine that executes Java bytecodes. The system is implemented in the Java programming language [2] and uses Java threads to implement several subsystems, such as the garbage collector [6] and the adaptive optimization system [4]. Figure 2 illustrates how Jikes RVM’s thread scheduler maps its M Java threads (application and VM) onto N Pthreads (user level POSIX threads). There is a 1-to-1 mapping from Pthreads to OS kernel threads. The operating system schedules the kernel threads on available processors. Typically, Jikes RVM creates a small number of Pthreads (on the order of one per physical processor). Each Pthread is called a *virtual processor* because it represents an execution resource that the virtual machine can use to execute Java threads.

To implement M -to- N threading, Jikes RVM uses compiler-supported quasi-preemptive scheduling by having the two compilers (baseline and optimizing) insert *yieldpoints* into method prologues, epilogues, and loop heads. The yieldpoint code sequence checks a flag on the virtual processor object; if the flag is set, then the yieldpoint invokes Jikes RVM’s thread scheduler. The flag can be set by a timer interrupt handler (signifying that the 10ms scheduling quantum has expired) or by some other system service (for example, the need to initiate a garbage collection) that needs to preempt the Java thread to schedule one of its own daemon threads.

This work builds on existing infrastructure [33] to capture hardware performance monitors, such as processor cycles, instructions completed, and L1 cache misses. The existing infrastructure generates a trace file for each Jikes RVM virtual processor, and one meta file. A trace file contains a series of trace records, which capture performance monitor

information for a measurement period in which exactly one Java thread was executing on that virtual processor. A trace record contains the following data:

Virtual Processor ID. This field contains the unique ID of the virtual processor that executed during the measurement period.

Thread ID. This field contains the unique ID of the Java thread that executed during the measurement period.

Thread Yield Status. This boolean field captures if a thread yielded before its scheduling quantum expired.

Real Time. This field contains the value of the PowerPC time base register at the start of this measurement period. The time base register contains a 64-bit unsigned quantity that is incremented periodically (at an implementation-defined interval) [23].

Real Time Duration. This field contains the duration of the measurement period using the time base register.

Thread Switch Status. The existing infrastructure collected a trace record only at thread switch time. Since our extensions can also generate a record for other reasons, we add a boolean field that describes what triggered the trace record creation.

Compiled Method IDs. The existing infrastructure captured the top two method IDs on the call stack. We extended this to capture the top $n \geq 0$ *compiled method IDs* on the call stack. A compiled method’s ID identifies a compiled method, which is a piece of machine code that has been created by compiling a method with a given compiler, at a given optimization level. The value of n is a runtime parameter to the VM.

Monitor Values. In the existing infrastructure this array contained the values of the selected hardware performance monitors, where the selection of monitors was a runtime parameter to the VM. In this work we generalize this to apply to software as well as hardware performance monitors. For a *counter* the captured value corresponds to the number of events that occurred during the measurement period.

Our new software performance monitors are implemented on two levels: in native code and in Java code. In native code, each Pthread has its private array of software performance monitors. A pointer to this array is stored as Pthread-specific data. This way, instrumentations in native code can update the monitor of the currently running thread. On the Java level, we keep a reference to a Java array of software performance monitors in the VirtualProcessor object. Jikes RVM provides cheap access to the VirtualProcessor object of the virtual processor (Pthread) on which the current Java thread is scheduled.

The existing infrastructure creates one trace file for each virtual processor. The real time values in the trace records are used to merge together multiple trace files to accurately model concurrent events on multiple processors. The values are also used to detect when the measurement period has been shared with other non-VM Pthreads.

A meta file is generated in conjunction with a benchmark’s trace files. The meta file specifies the number of hardware

and software performance monitor values and the number of compiled method IDs captured from the call stack in each trace record, and provides the following mappings: monitor field number to event name, thread ID to thread name, method ID to method signature, type ID to type name, and compiled method ID to method ID and optimization level.

2.4 Performance Analysis

Given a set of traces that contain trace records that provide the performance monitor values for each time slice, we want to investigate performance phenomena and problems. This subsection briefly describes the concepts of our performance visualization and analysis tool, the *Performance Explorer*, which is described in more detail in previous work [33]. We also describe the correlation facilities we added to *Performance Explorer* for this work.

2.4.1 Metrics

Metrics are a concept we use for performance analysis. They compute values given the values of performance monitors. A metric computes a value given a sample. Each sample contains the values of all recorded hardware and software monitors. Each of those monitors has an associated metric (e.g., the *Cyc* metric produces the value of the *Cyc* monitor of the given sample). Besides those monitor metrics we also support computed metrics, which represent arbitrary arithmetic expressions involving other metrics and constant values. This allows the computation of the *InstCmpl/Cyc* metric, given the *InstCmpl* and *Cyc* monitors.

2.4.2 Sample Lists

We provide a flexible and interactive way to declaratively construct sample lists, i.e., a list of samples, using filters on sample attributes and metrics, and using set operations on other sample lists. We also allow the direct manual selection of specific samples to include or exclude from sample lists.

2.4.3 Statistics

We provide descriptive statistics, such as min, max, average, and median, and multivariate statistics, such as cross correlation coefficient and correlation matrix.

2.4.4 Visualizations

For manual investigations we present straightforward plots of a metric over time. In addition, we also provide scatter plots for plotting two metrics against each other. Furthermore we provide a matrix of scatter plots, for the pair-wise comparison of any number of metrics. This matrix is actually a correlation matrix, showing a scatter plot in addition to the correlation coefficient in each cell. Examples of such visualizations are given in subsequent sections.

3. METHODOLOGY

This section describes the platform and benchmarks used for our case studies, and it quantifies the overhead introduced by our vertical profiler. The goal of vertical profiling is to find cause-effect relations between performance phenomena. Since this approach depends on the instrumentation of code on various levels, we need to verify that the phenomena we observe using vertical profiling are not caused by our instrumentation. This section describes the two components to this verification: perturbation analysis and validation.

Benchmark	Production	Vertical Profiling	
compress	9.77	10.15	3.6%
db	22.42	23.85	6.4%
jack	13.38	14.41	7.8%
javac	19.14	20.57	7.5%
jess	8.23	8.64	5.0%
mpegaudio	8.52	9.69	13.8%
mtrt	7.64	7.99	4.6%
jbb	27.17	31.84	17.2%
hsql	19.19	19.39	1.1%
Average			7.4%

Table 2: Vertical Profiling Overhead

3.1 Platform

We run all experiments on a 4-processor IBM POWER4 [18] machine running the AIX 5.1 operating system. Our virtual machine is an extended version of the Jikes RVM CVS head as of January 19, 2004.

3.2 Benchmarks

Table 1 presents our benchmark suite, which consists of the SPECjvm98 suite [10], a modified version of SPECjbb2000 [9], which we refer to as *jbb*, and the HSQL server [28]. The upper part of the table contains the single-threaded benchmarks, whereas the last three rows contain the multi-threaded applications.

For the SPECjvm98 benchmarks we use the reference inputs (size 100). We use a modified version of SPECjbb2000 that is able to process a fixed number of transactions (instead of running for a predetermined amount of time). We run the HSQL database server (version 1.7.1) in in-memory mode, using a modified version of the JDBC Bench included with the HSQL distribution.

3.3 Overhead

This section demonstrates that our implementation of vertical profiling is fast enough to be useful. Table 2 summarizes vertical profiling overhead. For each benchmark, the best of ten runs was chosen. All times are in seconds. The *Production* column gives the total time needed to execute the benchmark without any vertical profiling. The two *Vertical Profiling* columns show the total time needed to execute the benchmark with vertical profiling (tracing both HPMS and SPMS), and the overhead as a percentage of the production run. The overhead for vertical profiling ranges from as little as 3.6% for *compress* to as much as 17.2% for *jbb*. For this measurement, all 148 software performance monitors were enabled. Often a user might be interested in only a small subset of the monitors, and would thus be able to reduce overhead even more.

3.4 Perturbation Analysis

Because vertical profiling adds instrumentation to the system, it can perturb the very behavior that it is trying to understand. For example, instrumentation for collecting data on a software performance monitor may change the cache behavior of the application. Our perturbation analysis assures that the data collection is not perturbing the behavior of interest.

The following structure provides the framework we will use for the perturbation analysis in each of our case studies:

Benchmark	Functionality	Threads	Reference Input
compress	File compressor/decompressor	1 main	5 iterations compressing and decompressing files 213x.tar (3.1MB), 209.tar (2.8MB), 239.tar (0.9MB), 211.tar (1.2MB), 202.tar (1.1MB)
db	Simple in-memory address database	1 main	perform operations in scr6 (221 adds, 292 deletes, 8 modifies, 15 finds, 67 sorts) on address database in db6 (1.1MB, 15,332 records)
jack	Parser generator	1 main	17 iterations of generating the parser for grammar Jack.jack (17kB, 17 productions)
javac	Java compiler	1 main	4 iterations of compiling JavaLex.java (1.8MB, containing 144 classes) with -O
jess	Inference system	1 main	solve wordgames.clp (12kB, containing 2 puzzles: "GERALD+DONALD=ROBERT" and "5 houses, 5 attributes, 25 constraints")
mpegaudio	MPEG Layer 3 audio decoder	1 main	decode track2.mp3 (3.2MB)
mtrt	Multithreaded raytracer	1 main 2 renderers	render scene time-test.model (0.3MB, 2 lights, 5 spheres, 1,407 polygons with totally 4,233 vertices) into a 200x200 pixel image (each renderer renders a 100x200 section)
jbb	In-memory 3-tier application server	1 main 2 warehouses	execute 120,000 transactions per warehouse
hsqldb	In-memory SQL database server	1 main 2 clients	execute 4*10,000 JDBC queries per client

Table 1: Benchmarks

HPM perturbation analysis.

End-to-end perturbation analysis of HPMs.

Because HPMs are implemented in the hardware they do not perturb the behavior of the application. However, our mechanism for recording HPMs at each thread switch may perturb behavior. To see if this is the case, we conduct an additional run that does not collect any data at each thread switch, but instead records the end-to-end difference between the HPM values at the end and beginning of the run. We compare the end-to-end differences with the aggregate HPM values that are recorded at each thread switch. If these two sets of values are close, we have some confidence that our mechanism for recording HPMs does not perturb behavior at the macro level. To minimize inter-run variations due to things outside of our control (e.g., context switches at the operating system level) we use five runs for each configuration and use the average of the runs. To enable us to distinguish perturbation from measurement noise, we also compute the standard deviations for the five runs.

Temporal impact of HPMs. While the end-to-end perturbation analysis gives us confidence that we have not perturbed the application behavior at the macro level, it does not tell us if we have perturbed behavior at the micro-level. For example, the original application may experience a fixed cache miss rate throughout its run whereas the run that records HPMs at Java thread switches may experience varying miss rates during its run. If both runs end up with the same number of total misses, our end-to-end perturbation analysis will not recognize that we have perturbed the appli-

cation. We use qualitative analysis based on our knowledge of our HPM implementation to argue that vertical profiling is not perturbing the micro-level behavior of the application.

SPM perturbation analysis.

Impact of SPMs on HPMs. The instrumentation required to capture SPMs may also perturb system behavior. To verify that the mechanism for recording SPMs is not perturbing HPMs, we visually compare the HPM signals collected with and without SPM tracing enabled. If the signals visually correlate then we have some confidence that SPMs are not changing HPMs in a significant way.

Impact of SPMs on SPMs. We argue from knowledge of our SPM implementation that one SPM does not significantly perturb another SPM of interest to our case study.

3.5 Validation

A vertical profiling session results in a hypothesis about the cause of the observed performance phenomenon. In our studies we validate these hypotheses by eliminating the cause and running an additional experiment to verify that the phenomenon is gone. Each case study in Section 4 has a *Validation* subsection presenting the verification of our hypothesis.

4. CASE STUDIES

So far, the paper has claimed that vertical profiling is necessary for understanding the performance of Java programs. In this section we provide evidence for this claim by describing five case studies. Each case study shows how we used our infrastructure to explain a performance anomaly in a

benchmark program. We selected anomalies by gathering the IPC (Instructions completed Per Cycle) of our benchmarks over time (see Figure 3), and choosing the most significant patterns.¹ We found that in all the case studies we needed profile information from more than one system layer (Figure 1) to explain the anomaly.

Figure 3 shows the IPC over time for our benchmark programs. In this figure the multithreaded benchmarks use only one worker thread. The number to the right of each graph gives the IPC for all worker and main thread time slices. We see that not only do different benchmarks have vastly different IPC (factor of 2.3 between lowest and highest), but also the IPC changes dramatically over each benchmark run (up to factor of 7.8 between lowest and highest time slice in a benchmark).

We have marked the four most distinct patterns in Figure 3 as *sudden increase*, *gradual increase*, *dip before GC*, and *periodic pattern*. Four of our case studies use vertical profiling to explain these patterns. Our fifth case study uses vertical profiling to investigate the scalability of the three multithreaded benchmarks.

4.1 Gradual Increase in *Jbb*

From Figure 3 we see that the IPC of many benchmarks increases over time (*gradual increase* pattern). This section uses vertical profiling to explain the cause of this pattern in *jbb*. We picked *jbb* for this case study since the gradual increase pattern is particularly visible in *jbb* and moreover the pattern recurs several times during the run. For this case study we focus on the marked instance in Figure 3 which happens when the worker thread starts running (after the main thread is done setting up the application).

4.1.1 Background

The worker thread of *jbb* performs about 50 transactions per 10 ms time slice. Our vertical profiler reports performance monitor values at the end of each time slice. The changes in application behavior during a transaction are not captured in the signal, because our sampling rate is too low to observe individual transactions. Thus, we expect that the gradual increase in IPC is not due to the transaction behavior, but due to some other lower-level behavior.

In prior work [33] we speculated that *jbb*'s gradual increase in IPC occurs because as *jbb* runs, more and more of its code gets optimized. When a method runs for the first time, Jikes RVM uses its baseline (non-optimizing) compiler to compile the code; when a method becomes hot, Jikes RVM uses its optimizing compiler to recompile the code.

We came to the above conclusion in prior work because of two observations:

- We found that optimized code had a higher IPC (by more than 32%) than unoptimized code. Thus one could expect that the IPC of the application would increase with an increase in the amount of executed optimized code.
- We found that the number of flushes in the load/store unit (LSU) went down when the IPC increased. LSU

¹Although IPC does not necessarily reflect the amount of *useful* work completed (e.g. `noop` instructions completed are also counted) it is a popular metric used to measure CPU utilization of an application running on a particular software stack.

flushes happen when the POWER4's LSU speculatively reorders a load and a store that access the same address. The optimizing compiler uses a register allocator to reduce the number of loads and stores while the non-optimizing compiler simulates the Java expression stack, and issues frequent loads and stores to the top of the stack. If the POWER4 misspeculates the frequent loads and stores issued by unoptimized code, it incurs LSU flushes. We found that LSU flushes happen at 15.2% of the instructions completed for unoptimized code and 0.1% of the instructions completed for optimized code.

4.1.2 Findings

We were able to demonstrate that the gradual increase in IPC for *jbb* is indeed caused by the Jikes RVM optimizing more methods as the run progresses.

4.1.3 Exploration

Our prior work, which used only hardware performance monitors, hypothesized that the IPC for *jbb* increased gradually over time because as the program ran, more and more of the code got optimized. However, using only the hardware performance monitors we could not confirm this hypothesis.

To test this hypothesis, we needed to measure the amount of time spent in optimized code and unoptimized code for each time slice. Measuring this information directly would add overhead (and perturbation) to every call and return and thus we settled on an indirect way of capturing the information. We identified JVM lock acquisition performance monitors that would approximate this information. The Java compiler issues `MonitorEnter` bytecode instructions at each call to a synchronized method and at each entry into a synchronized block. Jikes RVM's baseline compiler expands `MonitorEnter` into a call to a lock acquisition method and the optimizing compiler expands `MonitorEnter` into an inlined body of a different lock acquisition method. We added two JVM performance monitors: *UnoptMonitorEnter* and *OptMonitorEnter*. *UnoptMonitorEnter* is incremented in the lock acquisition method used by baseline compiled code, while *OptMonitorEnter* is incremented by the inlined lock acquisition method used in optimized code. These JVM performance monitors told us how many `MonitorEnters` were executed in unoptimized and in optimized code. Since *jbb* executes many synchronized methods throughout its execution, these counts provide us with useful information. For benchmarks that do not execute synchronized methods throughout their execution, we would need different monitors to determine the time spent in optimized and unoptimized code.

Figure 4 shows the *IPC*, *LsuFlush/Cyc*, and the percentage of `MonitorEnter` byte code instructions executed that were unoptimized², over time. We can see that in the beginning the IPC is low, while the two other metrics are high. Over time the IPC increases while the other metrics decrease. The *LsuFlush/Cyc* and the percentage of *UnoptMonitorEnters* decrease at almost the same rate.

We use the correlation feature of the Performance Explorer to determine the cross correlation coefficient between these signals. Figure 5 shows two scatter plots. The left scatter plot correlates *Cyc/InstCmpl*³ and *LsuFlush/Cyc*. The

² $UnoptMonitorEnter / (UnoptMonitorEnter + OptMonitorEnter)$.

³We use the CPI, inverse of IPC, because the correlation

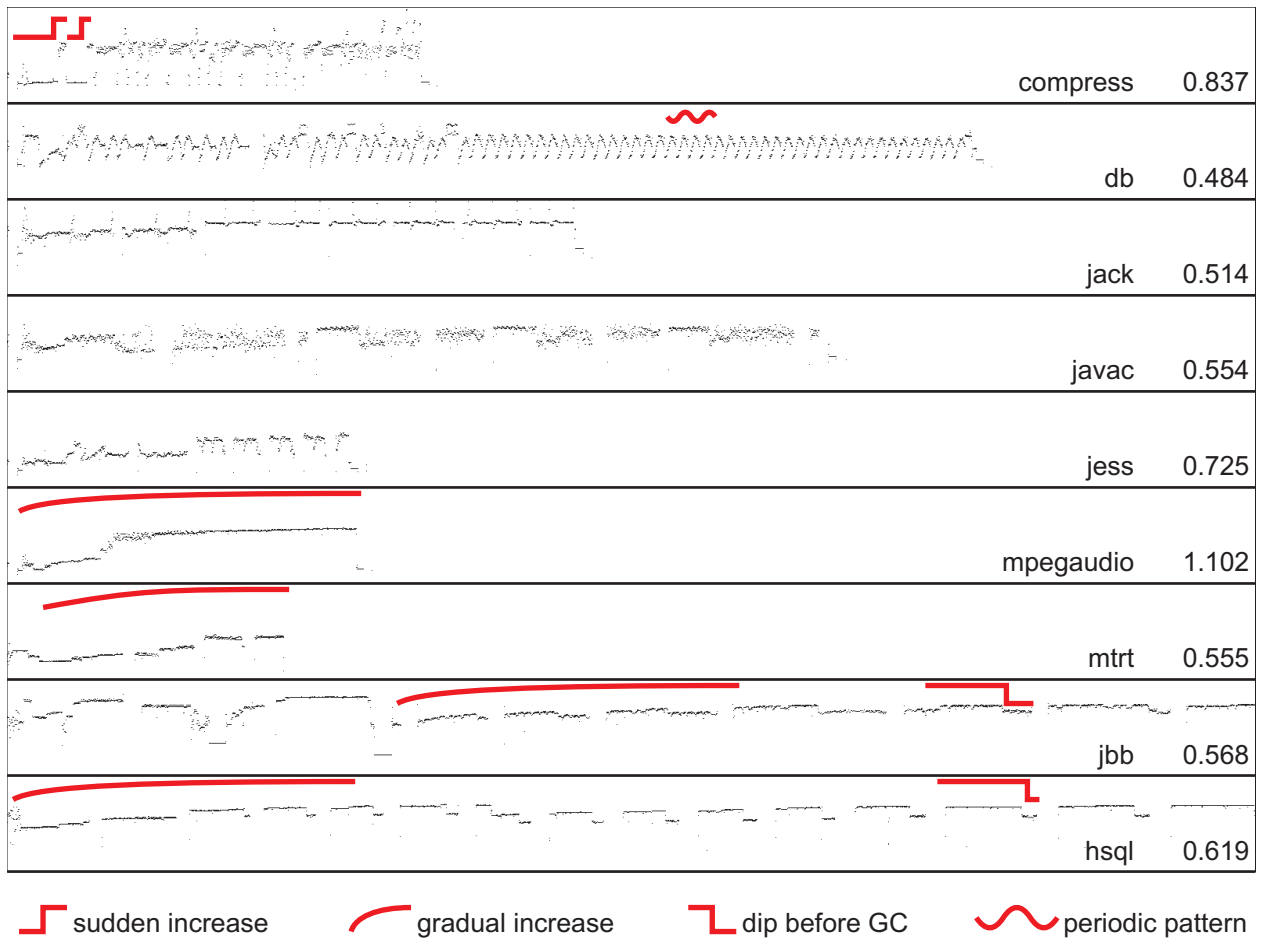


Figure 3: InstCmpl/Cyc over Time for All Benchmarks

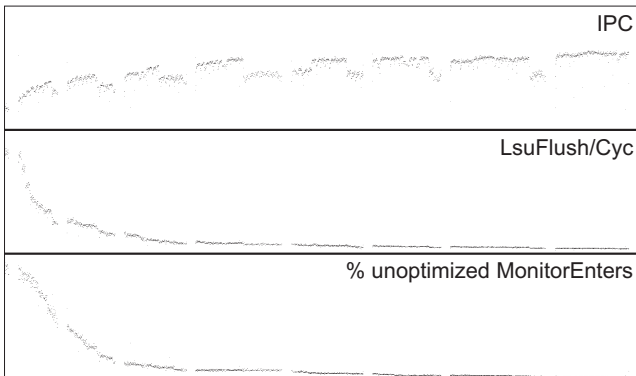


Figure 4: Behavior of *Jbb*'s Worker Thread over Time

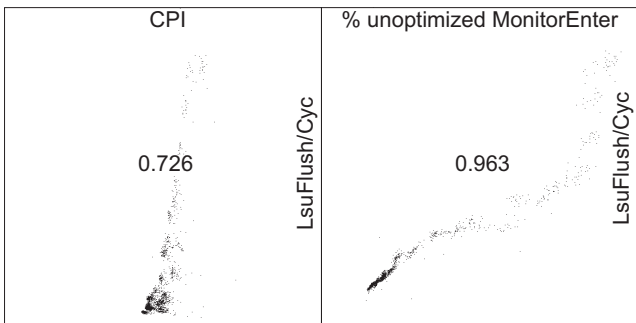


Figure 5: Correlation of Metrics for *Jbb*'s Worker Thread

plot shows that overall higher flush rates come with higher CPI rates. Also, the cross correlation coefficient of 0.726 indicates a significant correlation, which confirms the visual correlation. The right scatter plot correlates the *LsuFlush/Cyc* with the percentage of unoptimized *MonitorEnters*. Since the data points in this plot almost form a straight line⁴ it must be the case that the execution of unoptimized code is strongly correlated to *LsuFlush/Cyc* (correlation coefficient 0.963), which in turn is correlated to *Cyc/InstCmpl*.

Thus, our investigation confirms our hypothesis: the IPC of *jbb* increase gradually over time as more and more of the code it executes is optimized.

4.1.4 Perturbation Analysis

This case study examines the *Cyc*, *InstCmpl*, and *LsuFlush* HPMs. The end-to-end impact of HPMs is within measurement noise with the three HPMs decreasing with aggregate HPMs enabled: -0.92% for *Cyc*, -0.26% for *InstCmpl/Cyc*, and -2.57% for *LsuFlush*. Their standard deviations for five runs without vertical profiling enabled are 0.73% for *Cyc*, 0.63% for *InstCmpl*, and 1.48% for *LsuFlush*. It is highly unlikely that the huge gradual drop in *LsuFlush* over the

coefficient indicates a linear relationship between two metrics, and we expect a linear relationship between the overall cost of executing an instruction, CPI (*Cyc/InstCmpl*), and the amount of load store unit flushing, *LsuFlush/Cyc*.

⁴The curved segment in the right half of the graph consists of a small fraction of the points. The correlation coefficient is dominated by the vast majority of the points forming a straight line in the lower left corner.

course of the whole run is caused by HPMs as the HPM perturbation occurs consistently throughout the run at each Java thread switch. We were able to visually correlate the gradual increase in *IPC* and drop in *LsuFlush/Cyc* with and without SPM updating. The *UnoptMonitorEnter* and *OptMonitorEnter* SPMs can not perturb each other.

We conclude that the perturbation by our instrumentations is not significant enough to perturb our findings.

4.1.5 Validation

We validated our explanation for the gradual increase pattern in *jbb* by disabling the adaptive optimization system (*aos*), i.e. we used either the baseline or optimizing compiler exclusively without any recompilation. If our explanation is correct then (i) at the beginning of the run the *aos* system has no methods optimized and thus its metrics (such as *IPC*) will be similar to the run with the baseline compiler; and (ii) at the end of the run the *aos* system has optimized all the hot methods and thus its metrics will be similar to the run with the optimizing compiler. We found both properties to be true, thus validating our explanation. As further evidence we found that the runs with the baseline and optimizing compilers did not exhibit the gradual increase pattern, indicating that the *aos* is the cause of the pattern.

4.2 Sudden Increase in *Compress*

At the start *compress*'s execution, we find a variation of the *gradual increase* pattern presented in Section 4.1. In this case, the *IPC* suddenly jumps from 0.3 to 1.0.

4.2.1 Background

The execution of *compress* is dominated by the alternating invocation of two long running methods: *compress* and *decompress*.

4.2.2 Findings

The two long-running methods in *compress* account for most of its execution time. Since these methods dominate the execution time of *compress*, optimizing them results in a jump in *IPC* (which as we discussed earlier is higher with optimized code than with unoptimized code).

4.2.3 Exploration

In our previous case study (Section 4.1) we found that when the amount of executed optimized code increases the *IPC* also increases. However, unlike the previous case study, the *IPC* increase in *compress* is not gradual. We suspected that the rapid increase in *IPC* may happen when a key method gets optimized.

To investigate this, we used two software performance monitors at the JVM level. The first monitor, *TopOfStackMethodId*, captured and recorded the activation records at the top of the call stack. The second monitor, *OptimizedMethodId*, recorded the identity of the most recently optimized method at the end of each sample.

The first monitor told us that most of the time during the run of *compress* either the *compress* or the *decompress* method is executing. When we superimposed the trace from this monitor with the *IPC* trace we found that when the *IPC* jumps up for the first time, the method *compress* is executing and when the *IPC* jumps for the second time, the method *decompress* is executing. Looking at the second

monitor we find that the first jump in IPC corresponds to the optimization of `compress` and the second to the optimization of `decompress`. These two pieces of information tell us that it is likely that the two sudden increases in IPC are caused by the optimization of the two methods that account for most of the execution of the benchmark.

4.2.4 Perturbation Analysis

Similar to the previous case study, this case study examines the *Cyc*, *InstCmpl*, and *LsuFlush* HPMs. For *compress*, the end-to-end perturbation of HPMs is less than the standard deviation and, therefore, within measurement noise. In particular, end-to-end perturbations are 0.68% for *Cyc*, 0.5% for *InstCmpl/Cyc*, and 0.4% for *LsuFlush*, which is less than the corresponding standard deviations of 1.47% for *Cyc*, 0.22% for *InstCmpl*, and 6.74% for *LsuFlush*. It is highly unlikely that the temporal impact of HPMs cause the isolated jump in *InstCmpl/Cyc*. We were able to visually correlate the sudden increase in IPC and drop in LSU flushes with and without SPM updating. The SPMs *TopOfStackMethodId*, and *OptimizedMethodId* have no impact on one another as they count different events.

We conclude that vertical profiling does not invalidate our hypothesis.

4.2.5 Validation

To validate our explanation we reran *compress* with OSR (on-stack replacement [17, 16]) disabled. OSR replaces an unoptimized version of a method with an optimized version *while the method is running*. When OSR is disabled we expect the first execution of both methods to complete before the IPC jumps up.

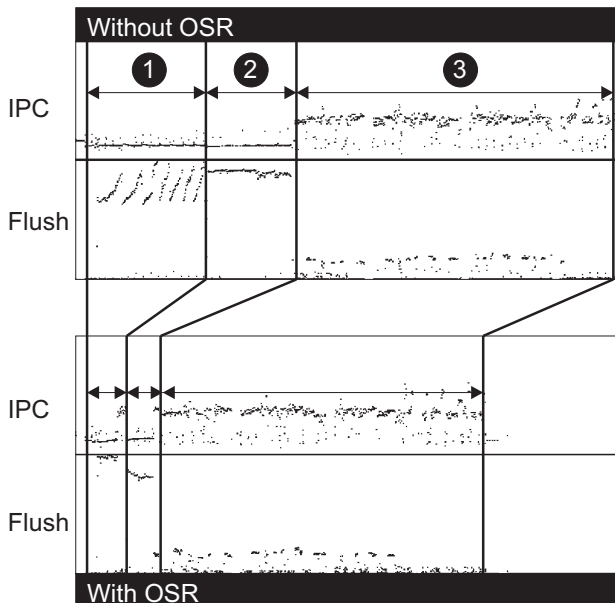


Figure 6: *InstCmpl/Cyc* and *LsuFlush/Cyc* for *Compress* with and without On-Stack Replacement

Figure 6 shows IPC and *LsuFlush/Cyc* over time for the two runs. The top two graphs are with OSR enabled and the bottom two graphs are with OSR disabled. The segment labeled 1 is the first execution of the `compress` method and the

segment labeled 2 is the first execution of the `decompress` method. We can see that the first invocations of `compress` and `decompress` take about 2.5 times longer without OSR than with OSR. This is because the baseline compiled methods cannot be replaced by optimized code until after their first invocation finishes. This data confirms our expectation and thus validates our explanation.

4.3 Scalability of Multithreaded Benchmarks

Three of our benchmarks, *mtrt*, *jbb*, and *hsql*, are multi-threaded applications. Each of them allows us to arbitrarily set the number of worker threads. When we first analyzed the traces of runs with more than one worker thread, we were surprised by the large number of time slices in those traces. We observed almost an order of magnitude more time slices (10,409 instead of 2,221 for *jbb*), and a much shorter average time slice duration, for a multithreaded run than for a single threaded run of the same benchmark with the same amount of work. This case study uses our vertical profiler to find the reason for this increase in the number of time slices.

4.3.1 Background

Each of the three benchmarks provides worker threads that can run in parallel. If we hand a fixed amount of work to a benchmark with two worker threads executing on a machine with two processors, we might expect the work to be done in about half the time it would take a benchmark with only one worker thread executing on one processor. This expectation is of course unrealistic because of the synchronization overhead associated with parallelization.

The fundamental synchronization feature in Java are critical sections. In Java a critical section corresponds to a synchronized block or a synchronized method. On entry to a critical section, Java executes the `MonitorEnter` bytecode. In Jikes RVM `MonitorEnter` is implemented as a call to the `VM.Thread.lock()` method. When calling `lock`, a thread either immediately gets the lock, retries a few times in a tight loop, or yields if neither of the former is successful. The number of yields happening from within the `lock` method are an indication of lock contention, and thus a measure of parallelization overhead.

In Java each object has an associated lock. A call to a synchronized instance method acquires the lock associated with the instance. A call to a synchronized class method acquires the lock associated with the `java.lang.Class` object describing the class. And an entry to a synchronized block acquires the lock associated with the object explicitly passed to the synchronized statement. Knowing the type (the Java class) of a contended lock's object can be very helpful for understanding the cause of lock contention in a parallel program.

4.3.2 Findings

We have found that a large number of time slices in multi-threaded runs is caused by lock contention. Worker threads try to acquire a lock that is already held by a different thread, and thus they yield. A thread that yields ends its time slice prematurely, and thus the length of the time slice is shorter than the scheduler's time quantum.

4.3.3 Exploration

Since our machine has four processors we run the benchmarks in the following configurations: 1 worker thread on 1 processor, 2 worker threads on 2 processors, and 4 worker threads on 4 processors. The only aspect that varies over the three experiments is the number of available processors and worker threads. The overall amount of work (*mtrt*: size of the picture; *jbb* and *hsql*: total number of transactions) stays the same. In the following analysis we focus on the behavior of the *worker threads* and we omit the system threads (compiler, garbage collector, ...) and the main thread (which generally just sets up the benchmark during the startup phase).

Table 3 presents our measurements for the three different levels of parallelism of the three multithreaded benchmarks. The first column shows the benchmark. The *Scale* column shows the level of parallelism (how many worker threads are executing on how many processors). The *Wall Time* column shows the wall clock time from the point where the first worker thread starts running to the point where the last worker thread stops running. The first subcolumn shows the time in million ticks (on the POWER4, a tick corresponds to 8 cycles). The second subcolumn shows the time as a percentage of the *1 on 1* time. The *CPU Time* column shows the sum of CPU ticks used by the worker threads. The *Samples* column shows the sum of the number of time slices used by all worker threads, and the *Sample D.* column shows the average duration (in million ticks) of the worker thread time slices. The last column gives the number of yields during lock acquisitions in the worker threads.

We expect the *wall time* to drop (the work to be completed sooner) with higher levels of parallelism, and the table shows that it generally does. The *CPU time* ideally would be constant over the various levels of parallelism. We find that the number increases, meaning that we actually use more CPU time for the same amount of work when we increase parallelism. Only *hsql* on 4 processors shows an increase in wall time over the 2 processor run. This is either because of the synchronization overhead growing very big, because of secondary effects (like decreased cache performance) caused by shorter time slices, or due to perturbation caused by incrementing our software performance counters.

We gather the number of *lock yields* using a software performance monitor, *LockYieldCount*. We can see that this value is 0 for all single-threaded runs. This is because there is no lock contention in this situation. For parallel runs, the value is greater than zero. We can also see that the number of time slices of a parallel run is almost equal to the number of time slices of the corresponding single threaded run, plus the number of lock yields in the parallel run. There is a slight difference because a thread can also yield in situations other than a lock acquisition. We currently cannot explain the big discrepancy (72,256 samples, but only 51,363 lock yields) observed in the *hsql* 4 on 4 run.

In the *hsql* 2 on 2 and 4 on 4 runs the time slices shrink to less than 1% of the scheduler’s scheduling quantum (of about 1.5 million ticks, or 10 ms). We expect such short time slices to negatively affect memory performance due to the frequent context switches. Our vertical profiler is the ideal environment for continuing the study in this direction, and we plan to do so in future work.

We have seen that the number of time slices goes up, and their length shrinks, as we increase parallelism, and that

Benchmark	Lock Yields	Library	VM	App
mtrt	0	0	0	0
mtrt	40	40	0	0
mtrt	350	134	216	0
jbb	0	1	0	0
jbb	2,704	0	2,703	1
jbb	8,013	113	7,843	57
hsql	0	0	0	0
hsql	28,600	2	0	28,598
hsql	72,012	143	149	71,720

Table 4: Classes Used for MonitorEnter

this change is correlated with the number of yields due to locking. Next we use another software performance monitor, *LockYieldTypeId*, to investigate which locks are causing the observed contention. This monitor is set to the type id (a unique integer identifier for each Java type) whenever a thread yields due to a lock acquisition (since it has to wait for the lock to be released by another thread).

Table 4 gives a breakdown for the lock yields, by the subsystem in which the lock type belongs. A *MonitorEnter* on a `java.lang.String`, for example, would show up in the *library* column. In this table we report all lock yields, not just the ones in the worker threads. As can be seen when comparing the *lock yields* column in this table with the same column in Table 3, the total number of lock yields is almost equal to the number of lock yields by the worker threads.

We can see that for *mtrt* contended locks are in the library and virtual machine classes. For *jbb* most contention is caused by the VM, whereas for *hsql* the contention is almost exclusively caused by the application. In *jbb*, the majority of the locks are associated with `com.ibm.JikesRVM.classloader.VM.NormalMethod`, and we suspect that this indicates contention in the runtime compilation subsystem. All of the 71,720 locks in *hsql* are associated with just one class, `org.hsqldb.Database`, which seems to be locked for a large part of the time, and thus causes a lot of contention.

Even though the primary goal of a vertical profiler is to correlate temporal behavior over different levels, this case study shows that it can also be used to analyze classical problems, like lock contention. We used two software performance monitors to gather the information we needed to not only find out that the short time slices are caused by lock yields, but also to further investigate what locks the application was yielding for.

4.3.4 Perturbation Analysis

Unlike the other four case studies, this study does not analyze a temporal pattern. Thus our perturbation analysis methodology is not fully applicable.

The SPMs never acquire any Java locks, nor do they cause a yield for any other reason. Thus they do not directly affect the duration of time slices, or the number of locks acquired. But the perturbation caused by the SPMs still is significant. It prohibited us from correlating the locking behavior to low level performance characteristics. In the future, we plan on reducing this perturbation by updating only the necessary monitors, and by reducing the overhead of a SPMs update.

Benchmark	Scale	Wall time (M,%)	CPU time (M,%)	Samples	Sample D. (M)	Lock Yields		
mtrt	1 on 1	1,070	100%	828	100%	575	1.440	0
mtrt	2 on 2	754	70%	957	116%	694	1.379	40
mtrt	4 on 4	666	62%	1,224	148%	1,129	1.084	302
jbb	1 on 1	4,276	100%	3,227	100%	2,221	1.453	0
jbb	2 on 2	2,401	56%	3,509	109%	4,979	0.705	2,703
jbb	4 on 4	1,346	31%	3,836	119%	10,409	0.369	7,940
hsq1	1 on 1	434	100%	265	100%	192	1.380	0
hsq1	2 on 2	313	72%	277	105%	28,849	0.010	28,600
hsq1	4 on 4	367	85%	317	120%	72,256	0.004	51,363

Table 3: Levels of Parallelism in Multithreaded Benchmarks

4.3.5 Validation

To validate our explanation we would need to change the amount of lock contention in the application. Since this requires significant changes to the application, we did not perform this validation and instead defer it for future work.

4.4 Dip before GC in *Hsql*

From Figure 3 we see that for many benchmarks the IPC dips immediately before a garbage collection (GC). GCs show up as gaps in the curves since GC runs in separate threads while all application threads are suspended, and Figure 3 presents only the data for the main and worker threads. In this study we analyze the pre-GC dip in *hsq1* in more detail. The *hsq1* run invokes GC fifteen times and for all except for the first two GCs, the IPC drops considerably, from about 0.65 to about 0.55, just before the GC.

4.4.1 Background

We first identified the pre-GC dip in prior work [33]. However, at that time we did not have our vertical profiling infrastructure and therefore were unable to find the cause for the dips.

We use a semispace GC in our configuration. The collector uses adaptive heap resizing, which means that it can increase or decrease the size of the heap based on the application’s allocation behavior. At startup the heap size is set to a user-specified initial value. When the application needs more space, the garbage collector grows the heap up to a user-specified maximum value. If the application needs less space than the heap size, the garbage collector can also shrink the heap. In this way an application takes only as much memory as it needs from the operating system.

4.4.2 Findings

We found the pre-GC dip to be caused by page faults to newly allocated pages. Since object allocation first uses existing memory before allocating from new memory obtained from the operating system, the dip happens immediately before a GC (which is triggered when the application runs out of memory).

4.4.3 Exploration

We started by using our tool to find metrics that correlated with the IPC. The top two graphs in Figure 7 show the IPC and a correlating OS-level metric, *EeOff/Cyc*. *EeOff/Cyc* gives the fraction of cycles when CPU exceptions are disabled (i.e., an interrupt handler is executing in the OS kernel). We see that exceptions are rarely disabled (0.2% of the cycles) except during the pre-GC dip, when exceptions

are disabled for 7% to 19% of the cycles.

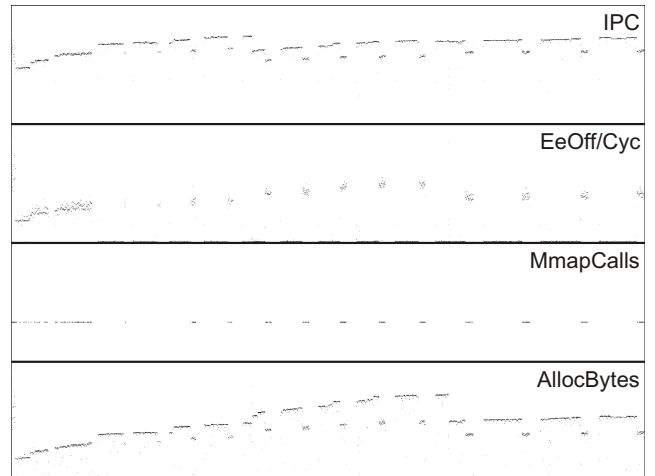


Figure 7: Behavior of *Hsql*’s Main and Worker Threads over Time

We initially thought that the pre-GC dips must be due to allocation patterns: after all, it is more likely that GC will be triggered in periods of intense allocation. To see if this was the case, we added the *AllocBytes* software performance monitor to capture the number of bytes allocated in Java.

We found that the rate of allocation was actually lower during dips than during normal behavior. Moreover, if the pre-GC dip was caused by a high rate of allocation, one would expect the dip to continue after the GC ended; we did not see that to be the case.

We continued our investigation to find the connection between garbage collection (on the JVM level) and exception handling (on the OS level). We added new performance monitors to record transitions from Java code to native code (the only ways through which the execution could invoke an OS system call). We found that one system call, *mmap*, correlated with the pre-GC dips. The third graph of Figure 7 gives data for the *MmapCalls* counter, which counts the number of *mmap* calls. We see that there are no calls to *mmap* except during the GC dip. Moreover, we found (using the *MmapBytes* monitor) that all calls to *mmap* map exactly 1 MB.

Table 5 summarizes the values of relevant performance monitors. Each row gives the data for one dip. The “GC” column identifies the GC for which the row reports pre-GC dip numbers. Even though GCs 1 and 2 do not have a visible

GC	AllocBytes(M)	MmapCalls	MmapBytes(M)	AllocB/MmapB	MmapPages	EeOff(M)	EeOff/MmapPages
1	44.2	45	45	0.98	11,520	298.3	25,894
2	41.7	42	42	0.99	10,752	272.9	25,381
3	3.3	3	3	1.11	768	22.9	29,818
4	5.4	5	5	1.09	1,280	35.2	27,500
5	7.8	8	8	0.98	2,048	48.9	23,877
6	8.6	9	9	0.95	2,304	53.9	23,394
7	9.3	9	9	1.03	2,304	59.7	25,911
8	10.9	10	10	1.09	2,560	70.2	27,422
9	11.3	11	11	1.02	2,816	72.5	25,746
10	11.1	11	11	1.01	2,816	73.2	25,994
11	11.4	11	11	1.04	2,816	70.9	25,178
12	10.8	11	11	0.98	2,816	69.9	24,822
13	10.9	11	11	0.99	2,816	70.9	25,178
14	11.2	11	11	1.01	2,816	73.0	25,923
15	11.3	12	12	0.95	3,072	75.4	24,544

Table 5: Information about Pre-GC Dips in Hsql

dip, we observe a considerable number of `mmap` calls before these GCs. Thus, we conclude that even though there is no visible dip before GCs 1 and 2, it is because the dip is much wider and in fact includes the entire execution before these GCs.

For GCs after the first two, we see that there are about 10 calls to `mmap` during each GC dip. From our visualizer we determined that `mmap` calls do not take longer than a single time slice, which corresponds to a single point in the IPC graphs. How, then, can 10 calls to `mmap` cause such a considerable dip in IPC? An `mmap` call occurs every fourth time slice during a dip but every time slice in a dip has a bad IPC.

We hypothesized that the pre-GC dips were related to adaptive heap resizing. When the garbage collector wishes to increase the size of the heap, it does not immediately go out and get the space. Instead, the allocator of the garbage collector obtains the space (via `mmap`) when the application actually needs it. Of course, once the space is allocated, the first accesses to pages in it will cause page faults. Those page faults are handled in an interrupt handler in the kernel, which has a considerably different IPC than the application. The pre-GC dip is much longer before the first two GCs because the application has not yet accessed any memory at the start of the run and thus all accesses before the second GC can potentially cause page faults.

perfectly correlated (correlation coefficient 0.9995), but the scatter plot shows that the application allocates almost exactly the number of bytes that are also `mmap`d during the dip. Thus, the JVM calls `mmap` to keep pace with the allocation requests of the application. Second, Table 5 shows that `MmapPages` and `EeOff` are probably also correlated. We verify this using the right scatter plot in Figure 8. We see that the number of cycles spent in the exception disabled state during a dip is linear in the number of `mmap`d pages (`MmapPages`)⁵. In fact we can see in column `EeOff/MmapPages` that we spend approximately 25,000 cycles for each `mmap`d page in an interrupt handler. Thus we conclude that the page faults due to `mmap`d pages cause the processor to execute a considerable amount of cycles in the page fault handler, which in turn impacts the IPC.

4.4.4 Perturbation Analysis

This case study examines the `Cyc`, `InstCmpl` and `EeOff` HPMS. The end-to-end perturbation for HPMS is within the standard deviation of the executions without HPMS. In particular, end-to-end perturbation is 0.23% for `Cyc`, 0.26% for `InstCmpl`, and 0.74% for `EeOff`, which is less than the standard deviation for executions without HPMS: 0.39% for `Cyc`, 0.41% for `InstCmpl`, and 1.02% for `EeOff`. Overhead caused by HPMS and preGC dips are not temporarily correlated. We were able to visually correlate the dips in IPC and bursts of `EeOff` before GC with and without SPMs. Finally, the `MmapCalls`, `MmapBytes` and `AllocBytes` SPMs do not impact each other.

We conclude that hypotheses that we derive from vertical profiling also hold for `hsql` without vertical profiling.

4.4.5 Validation

To validate our finding that the dip in IPC before GC is caused by page faults due to the `mmaps` needed for adaptive heap growth, we reran `hsql` with Jikes RVM’s adaptive heap resizing turned off. We found that the run without adaptive heap resizing did not exhibit the dips in IPC or the bursts of `mmaps`, thus validating our explanation.

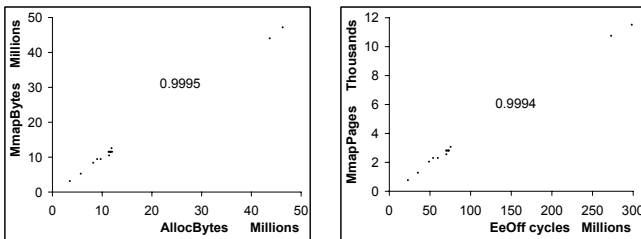


Figure 8: Correlation of Dips Before GC

Figure 8 provides evidence for our hypothesis. The left scatter plot illustrates the correlation between `AllocBytes` and `MmapBytes` over the 13 dips plus the full periods before the first two GCs. Not only are the two metrics almost

⁵The metric `MmapPages` is computed by dividing the `mmap`d bytes (`MmapBytes`) by the size of a page (4 kB).

4.5 Periodic Pattern in *Db*

From Figure 3 we see that *db* exhibits periodic behavior. The IPC of a typical period starts at 0.338, then rises to 0.568, and finally drops back to 0.338. Based on a visual inspection of the graph we counted more than 60 periods. This section uses vertical profiling to explain why *db* exhibits periodic behavior.

4.5.1 Background

db uses a `java.util.Vector` of `Entry` class instances to store its database. The `Entry` class has an instance variable of type `java.util.Vector` that stores the fields of the database record. Before performing a major operation on the database (e.g., sort or remove), *db* copies the vector that contains all the records of the database to an array.

db uses shell sort, an $O(n^2)$ algorithm, to sort the array representation of the database. Shell sort divides up the database into sets and uses insertion sort on each set. The first iteration of shell sort uses sets of size 2, the second uses sets of size 4, and so on, until the last iteration uses a set size that includes the entire database. The elements in a set are not adjacent in the database (except for the last iteration); instead, shell sort interleaves the elements of all the sets. For example, in the first iteration for a database of size n , the first set has elements at index 0 and $n/2 - 1$, the second set has elements at index 1 and $n/2$ etc.

4.5.2 Findings

Our exploration revealed that the periodic patterns in the IPC graph correspond to the 67 shell sorts in the *db* run. When the set size for the shell sort is small, the entire set fits in the L2 cache, yielding good IPC. Moreover, since the number of times shell sort touches each element increases with the size of the set, as long as the set fits in the cache, the IPC improves with increasing set size. When the set size exceeds 2178 entries, a set is too large to fit in the L2 cache, thus causing a drop in the IPC.

4.5.3 Exploration

Since the number of periods that we counted manually was the same as the number of times *db* sorts the database (Table 1) we immediately suspected that each period corresponded to a sort. We used the Performance Explorer to determine if there was a correlation between the IPC and any of the other existing performance monitors. We found an immediate high correlation (correlation coefficient -0.916) with a performance monitor at the processor level: L2 cache miss rate. We also determined that the L2 cache miss rate varied from 0.1 misses per 100 completed instructions at the troughs to 0.8 misses per 100 completed instructions at the peaks. Given that L2 cache misses are often expensive this swing in the L2 miss rate can account for the swing in the IPC.

While the processor-level performance monitor told us that L2 misses were responsible for the swings in the IPC, we still did not know what phenomenon actually *caused* the behavior. Moreover, we did not even know whether the troughs or the peaks signaled the start of a sort. To answer these questions, we added an application level performance monitor, `SetSize`, that keeps track of the set size of the shell sort. Figure 9 presents the signals for three metrics: IPC, $\log_2(\text{set size})$, and L2 cache miss rate. Rather than presenting the signals for the entire duration of the run, Figure 9 zooms in

on one part of the run.

We noted three interesting phenomena. First, once the set size exceeds about 2178, the IPC starts to drop. This phenomenon occurs because once the set size increases beyond 2178, it is too large to fit in the 1.5MB L2 cache. To access a single entry in the database, shell sort needs to perform six loads to entry specific data, with each load being to a different object (there are other loads also, for example, ones needed to perform array bounds checks but those are likely to have good locality in the cache and thus not relevant to this discussion): (i) load a reference to the entry object by indexing into the array of `Entry` instances; (ii) load a reference to the `Vector` inside the `Entry` instance; (iii) Load reference to the `Object` array inside `Vector`; (iv) load the element of the `Object` array to obtain a reference to the key on which to sort; (v) load the `Char` array that contains the contents of the key (which is a `String`); (vi) load contents of the key to use in the comparisons (this may actually be more than one load but most likely most of these loads will be to the same cache line). Load (i) is always to the same array, however, because of the nature of shell sort it is typically to elements that are far apart (e.g. element 0 followed by element $n/2 - 1$). Loads (ii) – (vi) are to distinct objects. In other words, there will be little spatial locality between two references to the same set. Given this and the 128-byte L2 line size on the POWER4, a 1.6MB⁶ L2 cache will be large enough to hold a set of size 2178 even if there is no spatial locality between two references to the same set. The next larger set size (which will be about twice the size) will not fit in the 1.5MB L2 cache. Thus, when shell sort increases the set size beyond 2178, its working set does not fit in the L2 cache, which degrades performance.

The second phenomenon we observe is that for set sizes ranging from 2 to 2178, the IPC increases as the set size increases. This phenomenon occurs because as the set size increases, shell sort has more temporal locality. For example, with a set size of 2, shell sort loads each element in the set only once. With a set size of 4, shell sort loads the first two elements thrice, the third element twice, and the fourth element once. Thus, as long as the set fits in the cache, the ever-increasing temporal locality leads to better performance.

The third phenomenon is that, contrary to what we expected, the shell sort does not start at either the trough or at the peak of the IPC curve. Instead it starts at a point that occurs immediately before the trough (circled in Figure 9). This happens because each sort is preceded by a phase that copies the contents of the vector that stores all the records in the database to an array. This phase has good locality because it reads sequentially from one data structure and writes sequentially into another data structure.

4.5.4 Perturbation Analysis

This case study examines the `Cyc`, `InstCmpl`, and `L2Misses` HPMS. The end-to-end perturbation for HPMS is either within or close to the standard deviation of the executions without HPMS. In particular, end-to-end perturbation is 2.77% for `Cyc` and 0.11% for `InstCmpl`, which is less than the standard deviation for executions without HPMS: 2.92% for `Cyc`, 0.23% for `InstCmpl`. Whereas, `L2Misses` HPM has only slightly more perturbation at 7.49% than its standard deviation at 5.97%. Tracing and periodic pattern are not

⁶ $2178 \cdot 6 \cdot 128 = 1,672,704$

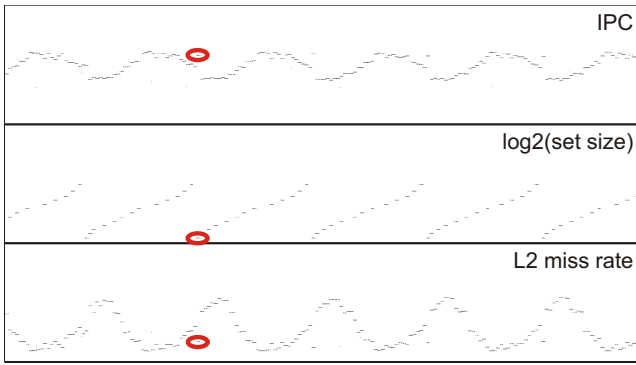


Figure 9: Behavior of *Db* over Time

correlated. We were able to visually correlate the *IPC* and *L2Misses* patterns with and without SPMs. The *SetSize* application SPM is not impacted by any other SPMs.

We conclude that vertical profiling does not invalidate the hypothesis that we derived.

4.5.5 Validation

We determined, using vertical profiling, that the *IPC* of a shell sort in *db* depends on whether the set it is sorting fits in the L2 cache. To validate this explanation, we manually performed object inlining (moving a referenced object into the referring object) on *db*'s main data structures. We expected that object inlining would enable larger sets to fit in the cache because it reduces the number of accesses to distinct objects in favor of more accesses to some objects. If object inlining *halves* the memory required to hold a set, then the *IPC* curve should start to drop at one higher set size than before.

After object inlining we found that for some of the sorts *IPC* did not start to drop until the set size reached about 5081 entries (as opposed to 2178 entries before). We believe that we did not see this phenomenon for all sorts because: (i) Object inlining reduced the memory required to hold a set by less than a factor of two. Since *db* performs insertions and deletions between sorts the set sizes used in two different sorts may be different and thus different sorts will have slightly different behavior; (ii) Our sampling frequency was not high enough to take at least one sample for every change in set size. In other words, if we did not have a sample at set size of 5081 we could not confirm that the *IPC* did indeed start to drop at 5081.

Overall, the object inlining optimization, guided by vertical profiling, decreased *db*'s run time by 66%. Part of this decrease is due to better cache locality (as discussed above) while part is due to a reduction in the number of instructions (after object inlining, one needs fewer pointer dereferences).

Thus, the above data supports our explanation for *db*'s behavior.

5. LESSONS LEARNED

Our five case studies exhibited different causes of significant performance impacts. We found that not only can performance be directly affected by the algorithms of the application, but also can other, less obvious causes, on different levels of the system, lead to significant performance changes. We have used a vertical profiler and a performance

analysis tool to find the causes of such performance phenomena. This section summarizes the lessons we have learned while applying our tools and techniques.

5.1 Layers

We mentioned in the introduction that for a full understanding of system behavior one needs to profile the system on multiple layers. Here we show what layers we actually had to observe for our case studies. Table 6 lists, for each case study, what monitors we used on what layer.

We can see that we needed information from multiple layers for each case study. For example, we could not have explained the *Gradual Increase* without the *LsuFlush* monitor that observes the hardware layer, and the *MonitorEnter* monitor observing the virtual machine layer.

The classification of a monitor into a layer depends on the context. In Table 6 we list the *EeOff* monitor in the operating system layer, even though *EeOff* is implemented as a hardware performance monitor. This is because we assigned monitors to layers based on what behavior they observe, not on the location of the instrumentation.

Table 6 also shows that we needed to use both, hardware and software performance monitors. It would not have been possible to observe the flushing of the LSU without the *LsuFlush hardware* performance monitor. And it would not have been possible to count the number of entries into synchronized code without the *MonitorEnter software* performance monitor.

But sometimes it is possible to monitor the same behavior with both, a hardware and a software monitor. For example the *EeOff*, measuring the time spent with CPU exceptions disabled, is equivalent to a software performance monitor that measures the time spent in interrupt handlers (e.g. by instrumenting the entry and exit of those handlers). In such a situation it is beneficial to use the hardware performance monitor, because it causes less perturbation.

5.2 Approach

We have been using two different but complementary techniques in our explorations of the causes of performance problems: *browsing* and *searching*.

5.2.1 Browsing

Whenever we were not sure about what the cause of the problem might be, browsing through the signals of all available performance metrics helped us find new hints. We thus depended on the set of available metrics to cover as many subsystems (and thus as many causes) as possible. In our previous work on using hardware performance counters to understand Java performance, browsing only through all available hardware performance metrics helped us find the initial hints for the *dip before GC* (the *EeOff/Cyc* metric) and the *gradual increase* (the *LsuFlush/Cyc* metric) patterns. The browsing process can be improved by semi-automatic support for detecting correlated signals, but there are some issues that prevent complete automation of this process (see Subsection 5.4).

5.2.2 Searching

Whenever we had a hypothesis, we needed to search for a performance metric we could use to test the hypothesis. Since we had a hypothesis, we knew what information we needed to test it. But often that information was not di-

Case Study Layer	Gradual Increase in <i>jbb</i>	Sudden Increase in <i>compress</i>	Scalability in <i>mtrt</i> , <i>jbb</i> , <i>hsql</i>	Dip Before GC in <i>hsql</i>	Periodic Pattern in <i>db</i>
Application					SetSize
Framework					
Java Libraries					
Virtual Machine	OptMonitorEnter UnoptMonitorEnter	TopOfStackMethodId OptimizedMethodId	LockYieldCount LockYieldTypeId	AllocBytes	
Native Libraries					
Operating System				EeOff MmapCalls MmapBytes	
Hardware	Cyc InstCmpl LsuFlush	Cyc InstCmpl	Cyc	Cyc InstCmpl	Cyc InstCmpl L2Misses

Table 6: Case Studies, Layers and Monitors

rectly available as a performance metric. Sometimes we could solve this problem by creating a computed metric (e.g. in our approximation of the percentage of the time spent in executing unoptimized code using MonitorEnter counters). At other times it was necessary to add a new software performance monitor (e.g. in observing the shell sort set size in Db). This requires an extensible system where the addition of a new counter is a simple and straightforward task. In contrast, there is no option for adding hardware performance counters, short of influencing the design of future microprocessors.

5.3 Perturbation

A vertical profiler needs to add instrumentations to the executed code in order to update its software performance monitors. Depending on the frequencies of those updates, the perturbation caused by these additional instructions can become significant. It is impossible to evaluate the perturbation outside the context of a specific exploration. There are too many possible combinations of software performance monitors that can be enabled and disabled. Furthermore, for some explorations there is a need to create additional application-specific monitors. We have found that an exploration based on a vertical profile needs to be backed by a perturbation analysis. Our approach is to first complete the exploration, without any concern for perturbation. Once we have found the root cause of a performance phenomenon, we then verify that the results have not been influenced by perturbation.

5.4 Correlation

The key issue in finding the cause of a performance phenomenon is correlation. This can be correlation between a performance metric and source code (e.g. program point X causes many i-cache misses), between performance metrics on different levels (e.g. high allocation rate leads to bad d-cache performance), or between performance metrics on the same level (e.g. high pipeline flush rate leads to bad IPC).

Correlation can be done visually (e.g. by comparing two time charts or by looking at a scatter plot) or statistically (e.g. by computing the cross correlation coefficient [30]). Statistical correlation requires a set of two dimensional data points, where each dimension corresponds to one metric. In our case studies we have correlated different entities (different kinds of data points): samples (time slices) and patterns.

In the *temporal correlation* we used for the gradual increase of IPC, we correlated two metrics across *all samples* in a sample list. In the *pattern correlation* we used for the dips before GC, we correlated two metrics across *all instances of a pattern* (e.g. pre-GC dip). Another option would be to do *benchmark correlation*, where we would correlate two metrics across *all benchmarks*.

We have identified several issues when using statistical correlation in our case studies. Sometimes the visual inspection of the two data sets would lead to the conclusion that they are highly correlated, even though the (linear) cross correlation coefficient is very small. Here we present a short summary of each of them.

5.4.1 Low Event Frequency

Given that we have a sequence of values for two metrics, we can apply a simple statistic such as Pearson’s cross correlation coefficient r to determine how well those two metrics correlate. This approach works well for metrics with event frequencies that are much higher than the sampling frequency used to generate the sequence of values. If the event frequency (e.g. for *SysMMapCalls*) is close to the sampling frequency, or even lower, then the two signals resemble those of Figure 10, and the cross correlation coefficient does not indicate any significant correlation.

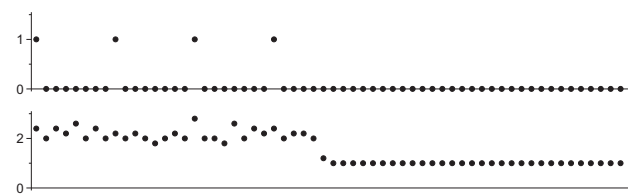


Figure 10: Correlation with Low Event Frequency ($r = 0.388$)

Figure 11 shows the distribution of event frequencies of the 240 performance counters that were incremented at least once in a *jbb* run with 120000 transactions and one warehouse thread. Each bar represents a performance counter (hardware or software), and its height represents the number of events that counter observed over the whole benchmark. The leftmost bar represents the *InstDisp* counter (67 billion instructions dispatched), and the second bar from the left

is the *Cyc* counter (46 billion cycles). We can see that we have counters with event frequencies distributed all over the frequency spectrum, from more than once per cycle, to once per 10 billion cycles. We also have 22 performance counters that were never incremented throughout this run (not shown in the figure).

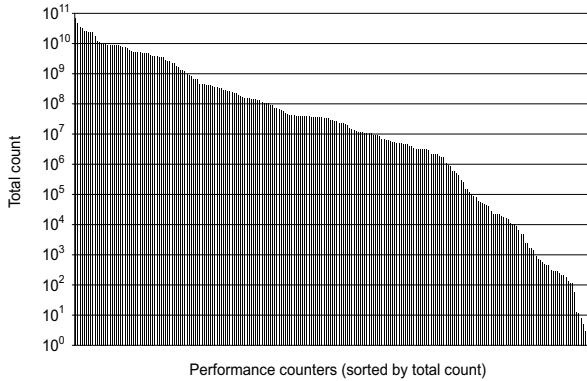


Figure 11: Overall Monitor Values of *Jbb*

Figure 12 shows the distribution of event frequencies for each of four levels of the system: hardware, operating system, native libraries, and virtual machine. We can see that on each level the total values of the monitors are distributed over a large range. Even on the VM level we can still observe monitors that represent very high frequency events, and even on the hardware level we can see monitors for very rare events.

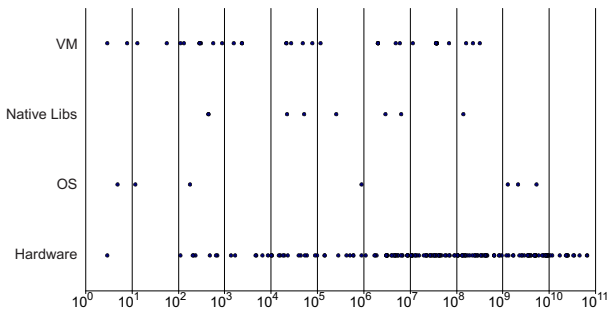


Figure 12: Overall Monitor Values of *Jbb*, by Level

5.4.2 Non-linear Relationships

The cross correlation coefficient is a measure of the linearity of a relationship between two signals. If the two signals are not linearly related (e.g. if one of them is an exponential function of the other), the cross correlation coefficient will potentially indicate no significant correlation. Thus it is important to correlate only metrics that have a mostly linear relationship. If the relationship between two metrics is suspected to be nonlinear, and if there is a hypothesis of what the function of the relationship is, then it can help to transform the metrics before correlation (e.g. to take the logarithm of one of the metrics).

5.4.3 Leverage Points

A further problem is the influence of extreme data points on the correlation coefficient. It is a well-known fact in statistics, that a single so-called *leverage point* [30] can highly influence correlation, as seen in Figure 13. The two sequences of values seem to be very well correlated, except for the last data point (the leverage point). But the correlation coefficient r is only 0.268. With the leverage point removed, the correlation coefficient increases to 0.929.

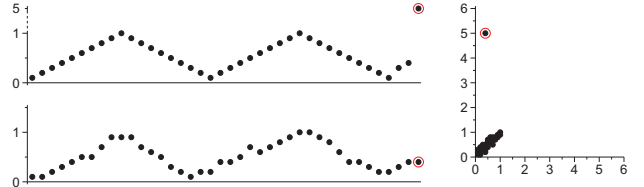


Figure 13: Correlation with Leverage Point ($r = 0.268$)

We have primarily found such leverage points due to the inconsistent duration of our samples. Even though Jikes RVM's scheduler preempts threads every 10 ms, some of our samples represent very long time slices (e.g. the garbage collection appears as one time slice, since it cannot be interrupted), while others represent very short time slices (some system threads that periodically wake up to perform a very small amount of work and then yield, or time slices in multi-threaded programs experiencing heavy lock contention and thus premature yields). In shorter samples an effect with a small absolute magnitude can have a large impact on a relative metric (see sub-subsection 5.4.5). Our solution was to filter out such time slices, if they significantly but inappropriately affected the correlation coefficient.

5.4.4 Direction of Causality

When investigating performance phenomena, we are generally interested in finding the cause of bad performance. Using our infrastructure, we can find out whether two signals are correlated. But the infrastructure does not tell us which signal is the cause, and which signal is the effect, or whether there is a causal relationship at all. We have observed cases where the IPC goes down when the cache miss rate goes up (which probably means that the frequent cache misses stalled the CPU). But we have also observed the opposite, where the IPC goes down and when the cache miss rate goes down (which probably means that something else caused the bad IPC, and the bad IPC caused fewer cache misses during the same amount of time).

We have found that in correlating the IPC with some other metric, the magnitude and the direction of the change in the other metric can indicate the direction of causality. If both IPC and the other metric change by about the same percentage and in the same direction, and the other metric is a measure of work, then this often indicates that the IPC is the cause for the change in the other metric (less work can be done because less instructions are completed per cycle). If the change in IPC is moderate, but the change in the other metric is extreme and in the opposite direction, then this can indicate that the other metric is the cause for the change in the IPC (an extreme amount of extra work had to be done, and thus less instructions could be completed per

cycle).

5.4.5 Absolute Magnitude of Metrics in a Ratio

It can be misleading to correlate a computed metric, like *LdMissL1/LdRefL1* (L1 data cache load miss rate), with some other metric. The miss rate could be very high, up to 1.0, but the number of load references might be so low (maybe just 1) that the influence on any other metric is negligible. Thus a high correlation between two metrics does not necessarily entail a cause-effect relationship.

6. RELATED WORK

This section surveys work related to vertical profiling. This includes work on gathering performance monitor data, profiling Java workloads, performance visualization, and statistical approaches to performance analysis.

6.1 Gathering Performance Monitor Data

Several library packages provide access to hardware performance monitor information, including the HPM toolkit [12], PAPI [7], PCL [5], and OProfile [27]. These libraries provide facilities to instrument programs, record hardware counter data, and analyze the results. The Digital Continuous Profiling Infrastructure provides a powerful set of tools to analyze and collect hardware performance counter data on Alpha processors [3]. VTune [36] and SpeedShop [39] are similar tools from Intel and SGI, respectively. Microsoft's Windows Management Instrumentation (WMI) [25] provides a framework for gathering software performance counter data over time. IBM's Performance Inspector [34] is a collection of profiling tools. It includes a patch to the Linux kernel providing a trace facility and hooks to record scheduling, interrupt, and other kernel events.

Our work differs in that we are gathering information about several levels of hardware and software simultaneously. We show in our case studies that this data is useful and often necessary for finding the root cause of a performance issue in a complex multilayered system. And we provide insight into the problems associated in correlating performance metrics across different levels.

6.2 Profiling Java Workloads

The Java Virtual Machine Profiler Interface (JVMPPI) defines a general purpose mechanism to obtain profile data from a Java VM [35]. JVMPPI supports CPU time profiling (using statistical sampling or code instrumentation) for threads and methods, heap profiling, and monitor contention profiling. Our work differs in that we are interested in infrastructure that is capable of measuring the architectural level performance of Java applications as well as the software level performance. Furthermore, our performance monitors can also measure effects that are not observable by using the JVMPPI.

Java middleware and server applications are an important class of emerging workloads. Existing research uses simulation and/or hardware performance counters to characterize these workloads. Cain et al. [8] evaluate the performance of a Java implementation of the TPC-W benchmark and compare the results to SPECweb99 and SPECjbb2000. Shuf et al. [32] analyze the memory performance of SPECjvm98 and pBOB on an IBM PowerPC processor using simulation and hardware performance counters. Luo and John [22] evaluate SPECjbb2000 and VolanoMark on a Pentium III pro-

cessor using the Intel hardware performance counters. Seshadri, John, and Mericas [31] use hardware performance counters to characterize the performance of SPECjbb2000 and VolanoMark running on two PowerPC architectures. Karlsson et al. [20] characterize the memory performance of Java server applications using real hardware and a simulator. They measure the performance of SPECjbb2000 and ECPerf on a 16-processor Sun Enterprise 6000 server. Other studies focus on behavior impacting specific subsystems, like Dieckmann et al. [13], who investigate memory performance metrics of interest for garbage collection designers. These studies generally focus on the overall characteristics of the workloads. We are interested in the causes of temporal performance phenomena, we present a system to gather the necessary information, and we introduce techniques for correlating performance information across different levels of a system.

Dufour et al. [14] introduce a set of architecture- and virtual machine-independent Java bytecode-level metrics for describing the dynamic characteristics of Java applications. Their metrics give an objective measure of aspects such as array or pointer intensiveness, degree of polymorphism, allocation density, degree of concurrency, and synchronization. Our work analyzes workload characteristics on the architectural level, and combines it with performance information from multiple software levels. While their work focuses on abstracting away from the hardware, we focus on connecting software behavior back to hardware performance.

6.3 Performance Visualization

A large body of work exists on performance visualization. Kimelman et al. [21] introduce PV, a performance visualizer focused on presenting temporal information from various levels of the system. PV shows only a subsection of the whole trace, but it allows scrolling through the whole trace, thereby continually updating the subsection currently visualized. Mellor-Crummey et al. [24] present HPCView, a performance visualization tool together with a toolkit to gather hardware performance counter traces. They use sampling to attribute performance events to instructions, and then hierarchically aggregate the counts, following the loop nesting structure of the program. Their focus is on attributing performance counts to source code areas. Miller et al. [26] present Paradyn, a performance measurement infrastructure for parallel and distributed programs. Paradyn uses dynamic instrumentation to count events or to time fragments of code. It can add or remove instrumentations on request, reducing the profiling overhead. Metrics in Paradyn correspond to everything that can be counted or timed through instrumentations. The original Paradyn does not support multithreading, but Xu et al. [38] introduce extensions to Paradyn to support the instrumentation of multithreaded applications. Zaki et al. [40] introduce an infrastructure to gather traces of message-passing programs running on parallel distributed systems. They describe Jumpshot, a trace visualization tool, which is capable of displaying traces of programs running on a large number of processors for a long time. They visualize different (possibly nested) program states, and communication activity between processes running on different nodes. The newer version by Wu et al. [37] is also capable of correctly tracing multithreaded programs. Pablo, introduced by Reed et al. [29], is another performance analysis infrastructure focusing on parallel dis-

tributed systems. It supports interactive source code instrumentation, provides data reduction through adaptively switching to aggregation when tracing becomes too expensive, and introduces the idea of clustering for trace data reduction. DeRose et al. [11] describe SvPablo (Source View Pablo), loosely based on the Pablo infrastructure, which supports both interactive and automatic software instrumentation and hardware performance counters, to gather aggregate performance data. They visualize this data for C and Fortran programs by attributing the metric values to specific source code lines.

Our work combines the virtues of these tools. We sample an extensive set of hardware performance monitors, combine this with software performance monitors injected at many levels of the software system, including in a virtual machine, and use our performance analysis tool to detect correlations, and ultimately cause-effect relations between performance phenomena across multiple levels.

6.4 Statistical Performance Analysis

Recent work uses statistical techniques to analyze performance counter data. Eeckhout et al. [15] analyze the hardware performance of Java programs. They use principal component analysis to reduce the dimensionality of the data from 34 performance counters to 4 principal components. Then they use hierarchical clustering to group workloads with similar behaviors. They gather only aggregate performance counts, and they divide all performance counter values by the number of clock cycles. Ahn and Vetter [1] hand-instrument several code regions in a set of applications. They gather data from 23 performance counters for three benchmarks on two different parallel machines with 16 and 68 nodes. Then they analyze that data using different clustering algorithms and factor analysis, focusing on parallelism and load balancing.

In our case studies we have found that the straightforward application of statistical techniques that rely on linear relationships between phenomena (such as cross correlation) did not help us in discovering cause-effect relationships in complex non-linear systems. We heavily relied on visualizations of the captured performance metrics over time, on identifying data points that were not related to the phenomena under investigation but due to their high leverage influenced the statistical computation, before verifying a correlation using statistics.

7. CONCLUSIONS

This paper introduces vertical profiling, an approach for understanding performance phenomena in modern systems. This approach captures behavioral information about multiple layers of a system and correlates that information to find the causes of performance phenomena. We have built an infrastructure to capture, visualize, and correlate vertical profiles which we have applied to five case studies. Each case study represents some performance anomaly that we explained using vertical profiling. The case studies demonstrate that vertical profiling is an effective method for understanding the behavior of Java applications.

To explain each performance anomaly, we used the following methodology. First, we performed a perturbation analysis to confirm that the performance anomaly actually existed. In other words, we confirmed that the performance anomaly was not an artifact of vertical profiling. Second, we

used our visualizer to find correlations (both visually and using statistical metrics) between the anomalous behavior and other metrics. Third, we followed the leads uncovered by the correlation to identify a potential explanation for the anomaly. Fourth, we performed a validation to make sure that we had indeed uncovered the correct reason for the performance anomaly.

In the future, we intend to look at ways to further automate our methodology.

To our knowledge this is the first work that has systematically used vertical profiling for performance analysis.

8. ACKNOWLEDGEMENTS

We thank Sam Guyer for the idea of using, and implementation of, object inlining to improve performance of db. We also thank Urs Hoelzle, Martin Hirzel, and Lauren Treacy for their feedback on the paper.

This work was supported in parts by the Defense Advanced Research Project Agency under contract NBCH30390004. Matthias Hauswirth and Amer Diwan were also supported by NSF ITR grant CCR-0085792, an NSF Career Award CCR-0133457, and an IBM faculty partnership award. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

9. REFERENCES

- [1] Dong H. Ahn and Jeffrey S. Vetter. Scalable analysis techniques for microprocessor performance counter metrics. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–16. IEEE Computer Society Press, 2002.
- [2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. *ACM SIGPLAN Notices*, 34(10):314–324, October 1999. Published as part of the proceedings of OOPSLA'99.
- [3] Jennifer M. Anderson, Lance M. Berg, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Sun tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: Where have all the cycles gone? *ACM Transactions on Computer Systems*, 15(4):357–390, November 1997.
- [4] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000. Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).
- [5] Rudolf Berrendorf, Heinz Ziegler, and Bernd Mohr. PCL - the performance counter library. <http://www.fz-juelich.de/zam/PCL>.
- [6] Stephen Blackburn, Perry Cheng, and Kathryn McKinley. Oil and Water? High performance garbage collection in Java with JMTk. In *26th International Conference on Software Engineering*, May 2004.
- [7] 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 2000 ACM/IEEE Conference on Supercomputing*, Dallas, TX, November 2000.
- [8] Harold W. Cain, Ravi Rajwar, Morris Marden, and Mikko H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Nuevo Leone, Mexico, January 2001.

- [9] Standard Performance Evaluation Corporation. SPECjbb2000 (Java Business Benchmark). <http://www.spec.org/jbb2000>.
- [10] Standard Performance Evaluation Corporation. Specjvm98 benchmarks. <http://www.spec.org/jvm98>.
- [11] Luiz DeRose and Daniel A. Reed. Svpablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the International Conference on Parallel Processing*, Fukushima, Japan, September 1999.
- [12] Luiz A. DeRose. The hardware performance monitor toolkit. In Rizos Sakellariou, John Keane, John Gurd, and Len Freeman, editors, *Proceedings of the 7th International Euro-Par Conference*, number 2150 in Lecture Notes in Computer Science, pages 122–131, Manchester, UK, August 2001. Springer-Verlag.
- [13] Sylvia Dieckmann and Urs Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer Verlag, June 1999.
- [14] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. Dynamic metrics for Java. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 149–168, 2003.
- [15] Lieven Eeckhout, Andy Georges, and Koen De Bosschere. How Java programs interact with virtual machines at the microarchitectural level. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 169–186, 2003.
- [16] Stephen J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization*, pages 241–252, 2003.
- [17] Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 32–43. ACM Press, 1992.
- [18] IBM. Power4 system microarchitecture. <http://www-1.ibm.com/servers/eserver/pseries/hardware/-whitepapers/power4.html>, 2001.
- [19] Jikes Research Virtual Machine (RVM). <http://www.ibm.com/developerworks/oss/jikesrvm>.
- [20] Martin Karlsson, Kevin E. Moore, Erik Hagersten, and David A. Wood. Memory system behavior of Java-based middleware. In *Proceedings of the Ninth International Symposium on High Performance Computer Architecture*, pages 217–228, Anaheim, CA, February 2003.
- [21] Doug Kimelman, Bryan Rosenburg, and Tova Roth. Strata-various: Multi-layer visualization of dynamics in software system behavior. In *Proceedings of the conference on Visualization (VIS'94)*, pages 172–178. IEEE Computer Society Press, October 1994.
- [22] Yue Luo and Lizy Kurian John. Workload characterization of multithreaded Java servers. In *Proceedings of the 2001 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 128–136, Tucson, AZ, November 2001.
- [23] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.
- [24] John Mellor-Crummey, Robert Fowler, and Gabriel Marin. HPCView: A tool for top-down analysis of node performance. In *Proceedings of the Los Alamos Computer Science Institute Second Annual Symposium*, Santa Fe, NM, October 2001.
- [25] Microsoft. Windows management instrumentation. http://msdn.microsoft.com/library/en-us/wmisdsk/wmi/wmi_start.page.asp, 2004.
- [26] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, 1995.
- [27] Oprofile. <http://oprofile.sourceforge.net>, 2003.
- [28] HSQL Database Engine (HSQLDB) Project. Hsql database engine. <http://hsqldb.sourceforge.net>.
- [29] Daniel A. Reed, Ruth. A. Aydt, Roger J. Noe, Philip C. Roth, Keith A. Shields, Bradley Schwartz, and Luis F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. In *Proceedings of the Scalable Parallel Libraries Conference*, October 1993.
- [30] Ashish Sen and Muni Srivastava. *Regression Analysis: Theory, Methods and Applications*. Springer-Verlag, 1997.
- [31] Pattabi Seshadri, Lizy John, and Alex Mericas. Workload characterization of Java server applications on two PowerPC processors. In *Proceedings of the Third Annual Austin Center for Advanced Studies Conference*, Austin, TX, February 2002.
- [32] Yefim Shuf, Mauricio J. Serrano, Manish Gupta, and Jaswinder Pal Singh. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205. ACM Press, 2001.
- [33] Peter F. Sweeney, Matthias Hauswirth, Brendon Cahoon, Perry Cheng, Amer Diwan, David Grove, and Michael Hind. Using hardware performance monitors to understand the behavior of Java applications. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium (VM'04)*, May 2004.
- [34] Bob Urquhart, Enio Pineda, Scott Jones, Frank Levine, Ron Cadima, Jimmy DeWitt, Theresa Halloran, and Aakash Parekh. Performance inspector. <http://www-124.ibm.com/developerworks/oss/pi>, 2004.
- [35] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, February 2000.
- [36] Intel VTune performance analyzers. <http://www.intel.com/software/products/vtune>.
- [37] C. Eric Wu, Anthony Bolmarcich, Marc Snir, David Wootton, Farid Parpia, Anthony Chan, Ewing Lusk, and William Gropp. From trace generation to visualization: A performance framework for distributed parallel systems. In *Proc. of SC2000: High Performance Networking and Computing*, November 2000.
- [38] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Principles Practice of Parallel Programming*, pages 49–59, 1999.
- [39] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS R10000 performance counters. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, November 1996.
- [40] Omer Zaki, Ewing Lusk, William Gropp, and Deborah Swider. Toward scalable performance visualization with Jumpshot. *High Performance Computing Applications*, 13(2):277–288, Fall 1999.