Vertical Profiling: Evaluating Computer Architectures using Commercial Applications

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

Matthias Hauswirth
University of Lugano, Switzerland
Matthias.Hauswirth@unisi.ch

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

ABSTRACT

This paper demonstrates how a performance analysis technique, vertical profiling, can be used to determine the cause of a performance anomaly: a gradual increase in instructions per cycle over time. Understanding the cause required trace-information from multiple layers of the execution stack (application, Java virtual machine, and hardware), expert knowledge in each layer, and repeated application of the process. To evaluate today's complex software and hardware systems requires sophisticated performance analysis techniques. Nevertheless, as future software and hardware systems become more complex, these performance analysis techniques must be automated.

1. INTRODUCTION

Using commercial applications to evaluate computer architectures has always been an onerous task, requiring both time and expertise. Time is required to identify which components of the application and the execution environment are affecting the underlying hardware. Expertise in a particular component is required to understand the component's affect on the underlying hardware. Nevertheless, evaluation and performance analysis is becoming more critical, because performance improvements are no longer available from hardware alone: due to power and thermal constraints, clock frequencies are not increasing at their previous rates of doubling every eighteen months.

Unlike scientific applications, commercial workloads have a complicated software stack. Consider the Java software stack that executes on top of a Java virtual machine, an operating system (OS) and hardware. The Java software stack typically consists of the application, one or more frameworks, an application server, and a plethora of libraries (application, framework, language and native). The complexity of this execution stack (software and hardware), and the interactions between the layers of the execution stack makes evaluating computer architectures difficult, because it is not always obvious what layer in the execution stack and what component in that layer is causing the effect on the underlying hardware. In contrast, the software stack for scientific applications is simpler, consisting of the application, perhaps a library, that sit on top of an OS and hardware.

Consider the Java virtual machine (JVM). Most JVM's have a number of components, including: a garbage collector (GC) that manages memory deallocation and supports the Java language's location transparency of data, a dynamic compiler that compiles Java code at runtime, and an adaptive optimization system (AOS) [2] that determines when to optimize a method and at what optimization level. These three components place different demands on the underlying hardware. Therefore, knowing that the JVM places demand on the underlying hardware is not sufficient unless the impact of the individual JVM components are teased apart.

Traditional approaches to evaluating computer architectures with commercial applications usually focus on only one layer in the execution stack. Although helpful, looking at only one layer in isolation does not provide the complete picture of performance. We have found that in many cases the temporal performance of a workload can only be understood by analyzing the interactions between layers of the execution stack [11]; that is, the interactions between the software stack, JVM, OS and hardware must be analyzed as a whole and not independently.

In addition, traditional approaches to performance analysis assume that metrics are "constant" when the application reaches its "steady-state". Our experience has shown that this is not true. As illustrated in Figure 1, a metric such as *IPC* (instructions per cycle) is rarely, if ever, flat for long periods of time. It is precisely a metric's change over time that we are interested in understanding to evaluate the underlying hardware.

This paper demonstrates how *vertical profiling* can be used to better understand the interactions between the various layers of the execution stack and the demands they place on the underlying hardware. Section 2 presents the complexity of the execution stack for commercial applications. Section 3 describes vertical profiling. Section 4 presents a case study illustrating how vertical profiling was used to analyze the temporal system performance. Section 5 discusses related work and why traditional tools are not sufficient to evaluate computer architectures with commercial applications. Section 6 presents conclusions. Section 7 discusses

This paper appeared and was presented at the Ninth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-9), February 12, 2006, Austin, Texas, USA.

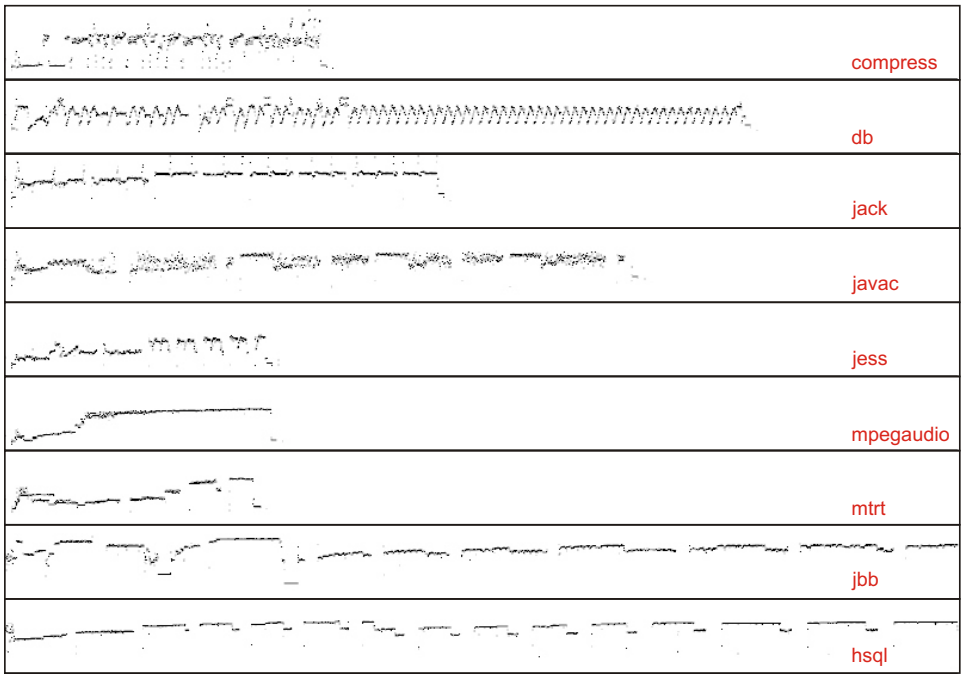

Figure 1: Instructions per cycle (IPC) over time for a collection of Java benchmarks. All signals were gathered on a PowerPC POWER4 processor running AIX 5.1.

future work.

2. MOTIVATION

Commercial applications are increasingly constructed by gluing together software taken from third party developed libraries and frameworks and executed on top of a complex runtime environment (JVM) that automatically provides services such as garbage collection and dynamic optimization. Although the third-party software typically has well-defined functional specifications, performance specifications are poorly-defined, if existent at all. In addition, how the Java software stack interacts with the JVM services is not always obvious or predictable by the application developer. Therefore, many performance problems occur from the interaction between the application and the independently developed software, and between the Java software stack and the JVM.

Consider the JVM and how an application interacts with three of its components: a *garbage collector* (GC) that manages memory deallocation and supports the Java language's location transparency of data; a *dynamic compiler* that compiles Java code at runtime; and an *adaptive optimization system* (AOS) [2, 3] that determines when a method should be compiled and at what level of optimization. Each component that is exercised by a Java application can have an effect on the underlying hardware. For instance, the allocation and referencing behavior of the application influences how the garbage collector traverses and reclaims objects, which, in turn, influences page faults and cache misses, which ultimately influences overall performance. Because of such interactions, the complexity of analyzing the performance of a modern application grows exponentially with the number of components exercised in the application.

Furthermore, individual components can have radically different impact on underlying hardware. Consider icache miss rates. For example, a garbage collector is typically implemented as a tight loop that fits into the icache and thus has low cache miss rates. In contrast, the AOS executes infrequently and, during each execution, executes for a short duration of time resulting in high cache miss rates.

The dynamic compiler has a range of icache miss rates that falls between the AOS and GC depending on the size of the method that it is optimizing and the level of optimization. Compiling a small method at a low level of optimization has a longer duration than a AOS thread's execution and thus will have lower icache miss rates due to temporal cache locality. Compiling a large method at a high level of optimization may span multiple scheduling quanta, during which the compiler will time share the processor with other Java threads that will compete for cache lines in the icache resulting in higher icache miss rates than the GC.

Therefore, understanding a JVM's impact on the icache miss rates requires that the individual components of the JVM layer are distinguished.

The task of analyzing the interaction between components becomes more difficult if those interactions change during an application's execution. For example, the interactions may change over time due to a phase shift in the application, or due to the dynamic compiler optimizing code.[1] Thus, capturing the temporal behavior of a modern system is crucial to understanding its performance. This temporal behavior can be modeled as a sequence of observed events, called a *trace*.

Therefore, knowing how layers in the execution stack and how components in those layers interact and affect the un-

[1]A dynamic compiler usually has multiple levels of optimization that may be applied successively to a method as the total time the method executes increases.

derlying hardware requires trace information across the layers of the execution stack and requires that the information is differentiated by each layer's components.

3. VERTICAL PROFILIING

This section provides background information on *vertical profiling* [11, 9], a performance analysis technique that helps to identify which layer in the execution stack and which component within that layer is affecting the performance of a computer architecture. Vertical profiling uses traces of metrics from different components of the different layers of the execution stack. Examples of metrics are:

Instructions per cycle. Hardware layer, overall performance.

Cache misses. Hardware layer, memory hierarchy component.

Page faults. OS layer, virtual memory management component.

Memory map calls. OS layer, virtual memory management component.

Compilations at optimization level 3. JVM layer, dynamic compiler component.

Bytes allocated. JVM layer, memory allocation component.

Transactions completed per minute. Application layer.

The trace for a particular metric is called a *signal* and can be plotted as a graph where the x-axis is time and the y-axis represents the metric's value. We typically aggregate metrics over time intervals. For Java applications, we aggregated metric values at every Java thread switch where the scheduling time quantum is 10 milliseconds [11, 15].[2] Each point in a signal represents the metric's aggregate value for that time interval. Figure 2 illustrates three such signals for a Java application.

Our experience has taught us that different Java threads and JVM services have different performance characteristics, and that partitioning signals by Java threads and JVM services is important for evaluating computer architectures.

Given a set of signals, vertical profiling consists of five steps:

1. Identify a performance anomaly as an interval in a target metric's signal. For example, a change in a metric's value over time,
2. Correlate the corresponding intervals in the signals of other metrics with the performance anomaly.
3. Identify a causal metric that correlated highly with the target metric.
4. Validate the causal metric.
5. Iterate using a causal metric as the performance anomaly.

Because correlation does not always imply causality, the third step requires reasoning to eliminate metrics that have accidental correlation with the target metric. For example, if the number of instructions that complete per cycle goes down, the number of transactions that complete per minute would be expected to go down proportionally, but the reduction in transactions completing per minute would not be the cause for the drop in *IPC*.

A vertical profiling study can require the collection of a large number of metrics. It is often impractical or even impossible to collect all these metrics in a single execution of the system for the following reasons:

- Not all metrics can be captured together. For example, the hardware performance monitors typically have a small number of registers (on the order of 10) that count the number of times an event occurs; however, modern processors have hundreds of events.
- Even if all the metrics could be captured together, the perturbation to the system would be prohibitive.
- Finally, performance analysis is an iterative process: signals for metrics may not be collected at the same time.

Nevertheless, reasoning across traces is nontrivial, because even two runs of a deterministic application using hte same inputs yields slightly different behavior due to non-determinism in the OS an JVM. We have applied a technique from speech recognition, dynamic time warping (DTW), to align traces. Aligning several traces allows us to correlate metrics that had to be collected in different executions of the system. Although initial results [10] have found that DTW works well for aligning traces, we continue to do research in this area.

4. USING VERTICAL PROFILING

In this section, we use vertical profiling to understand how a commercial application and its execution environment affect the underlying computer architecture. The representative commercial application that this section uses is *jbb*, a modified version of the SPECjbb2000 [7] (Java Business Benchmark) that evaluates the performance of server-side Java by modeling a three-tier application server. For benchmarking purposes, *jbb* was modified to run for a fixed number of transactions (120,000 for this experiment) instead of running for a fixed amount of time. All of our experiments were performed on a POWERPC POWER4 1.2 GHz, 64-bit microprocessor running AIX 5.1.

When we looked at the *IPC* signal for *jbb*, we noticed that the *IPC*'s value gradually increases over time. The top graph in Figure 2 illustrates this phenomena, where the x-axis is time and the y-axis is the *IPC* value. The signal in the top graph of Figure 2 is a prefix of the worker thread that starts after the main thread.[3] We use the behavior of this signal as a performance anomaly and use vertical profiling to determine its cause.

[2]For C and Fortran scientific programs, metric values were aggregated at 10 millisecond intervals defined by timer interrupts [5].

[3]In addition to being an interval, the *IPC* signal in the top graph of Figure 2 differs from the *IPC* signal for *jbb* in Figure 1 (the eighth graph from the top), because we have eliminated another performance anomaly, dip before GC, that was also resolved using vertical profiling [11]. A GC is represented as a gap in the signal when the GC is runing and, in Figure 1, the dip before GC can be seen as the segments of the signal that decrease just before the gaps.

4.1 Investigation

Given the performance anomaly, we first correlated the hundreds of POWER4 hardware performance monitor (HPM) signals with the performance anomaly [15] to identify the highly correlated signals as potential candidates for the cause of the performance anomaly. Although there were a number of potential candidates, the load/store unit flushes per cycle (*LsuFlush/Cyc*) signal stood out as a possible cause. The other potential candidates were determined, by examination and reasoning, to not be a cause of the performance anomaly. For example, loads per cycle correlated highly with *IPC*, however, a reduction in the number of loads does not cause a reduction in *IPC*, because loads are a subset of all instruction. As pointed out in Section 3, correlation may not imply causality. In vertical profiling, correlation between two signals provides the starting point to look for causality.

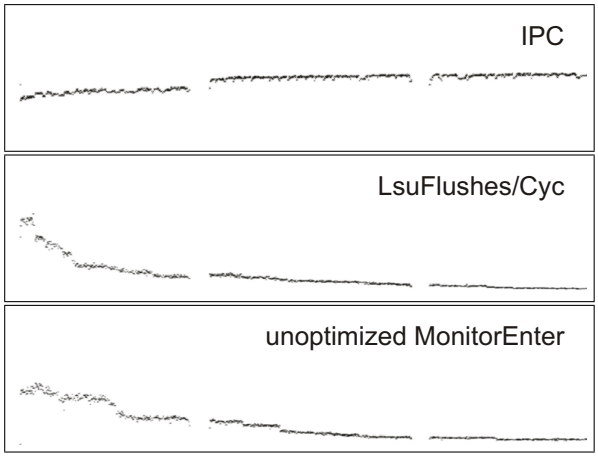

Figure 2: Behavior of *jbb*'s worker thread over time.

The top two graphs in Figure 2 illustrate a visual correlation between the *IPC* and the *LsuFlush/Cyc* signals: as *IPC* increases, *LsuFlush/Cyc* decreases. The left scatter plot of Figure 3 correlates *LsuFlush/Cyc* with *CPI* cycles per instruction, the inverse of *IPC*. The plot shows that overall higher flush rates correlates with higher *CPI* rates. Furthermore, the cross correlation coefficient of 0.726 indicates a significant correlation, which confirms the visual correlation.

Talking to hardware experts, we learned that a pipeline flush happens when the POWER4 speculatively reorders two data dependent memory access instructions. A POWER4

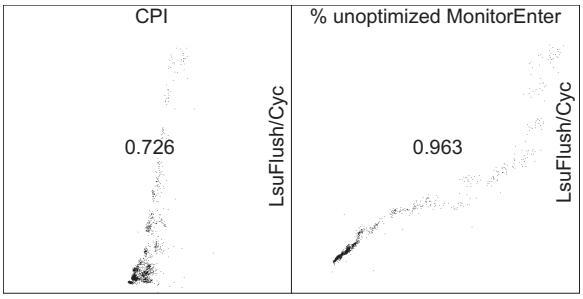

Figure 3: Correlation of *jbb*'s worker thread's signals.

Metric	Baseline	Opt 0
IPC	0.348	0.508
LsuFlush/Cyc	0.053	0.001

Table 1: The aggregate measurement of instructions per cycle (IPC) and load/store unit flushes per cycle (LsuFlush/Cyc) for the baseline and optimization level 0 JIT compilers.

is a speculative, out-of-order superscalar machine that has two load/store functional units and issues instructions in groups of up to five instructions. Intuitively, at each cycle, hardware looks in parallel in each load/store units' issue queue for a memory instruction to execute. If two memory access instructions are data dependent, but are issued on different load/store units, the hardware may speculatively execute the dependent instruction and violate the data dependency causing a misspeculation. When a misspeculation is detected, the hardware flushes the instruction pipeline, and reschedules the two dependent instructions to the same load/store issue queue to prevent the misspeculation. Nevertheless, flushing the pipeline is an expensive operation.

4.2 Iterate

Although the *LsuFlush/Cyc* metric explains the cause of the performance anomaly at the hardware layer, only looking at hardware layer metrics provided no insight into which component in the software stack is causing the flushes.

Our first instinct was to look at metrics at the application layer to determine if the application's behavior changed over time. In particular, we looked at the mix of transactions to determine if the performance anomaly was due to transactional behavior. With an application layer metric, we confirmed that the mix of transactions over time was relatively fixed and did not correlate with *IPC*. Thus, we concluded that the performance anomaly was not due to the transaction behavior, but due to some other lower-level behavior.

Using our Java virtual machine (JVM) expertise, we next focused on the JVM layer of the software stack, and within this layer the adaptive optimization system (AOS). The JVM that we used was Jikes RVM [12], an open source, research virtual machine. The AOS monitors an application's runtime behavior and recompiles methods over time at higher levels of optimization as the amount of execution time spent in that method increases. The first time a method is executed, the AOS compiles the method at an unoptimized baseline level of optimization. Baseline compilation ensures fast compilation times, but sacrifices code quality. For example, baseline compiled code explicitly models the Java byte code representation of an expression stack as loads and stores to memory. All other higher levels of optimization, use register allocation to keep the expressions in registers whenever possible, instead of memory.

Jikes RVM supports a JIT (just-in-time) compiler configuration where the first time a method is executed, it is compiled at a particular optimization level and never compiled again. Table 1 presents the difference in *IPC* and *LsuFlush/Cyc* when using the baseline (Baseline) and optimization level 0 (Opt 0) JIT's for one of the SPECjvm benchmarks. These values are aggregated over the complete

execution of the benchmark. The important thing to notice is that *IPC* for the optimization level 0 is 46% higher than the baseline and that the number of *LsuFlush/Cyc* is almost nonexistent for the optimization level 0, but significant for the baseline: one in every 20 cycles is involved in flushing the pipeline! With this insight, our hypothesis is that the AOS behavior of initially compiling methods at baseline and only over time recompiling them at higher levels of optimization was causing the initial high number of pipeline flushes that then decreased. Although the aggregate information illustrates the impact of different optimization levels on *IPC* and *LsuFlush/Cyc*, the information provides no insight into whether there is a high correlation between the amount of optimized code that is executed and *IPC*. The correlation is needed to prove our hypothesis.

To test this hypothesis, we measured the amount of time spent in optimized code and unoptimized code over time. 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. Instead, we identified a JVM lock acquisition metric that approximated 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 metrics: *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 metrics told us how many `MonitorEnters` were executed in unoptimized and in optimized code. Because *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, other metrics would be needed to determine the time spent in optimized and unoptimized code.

The bottom two graphs in Figure 2 illustrate a visual correlation between the *LSUFlush/Cyc* and *UnoptMonitorEnter* signals: as the *LSUFlush/Cyc* signal decreases so does the *UnoptMonitorEnter* signal. The right scatter plot of Figure 3 correlates the *LsuFlush/Cyc* with the percentage of unoptimized `MonitorEnters`. Because the data points in this plot almost form a straight line it seems highly likely 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 determined that the performance anomaly in *jbb*, *IPC* increases gradually over time, is caused by the AOS: over time as the AOS optimizes more and more methods the optimized code reduces pipeline flushes and improves *IPC*.

4.3 Validation

We validated our explanation for cause of the performance anomaly by using Jikes RVM's JIT configurations to disable the AOS and to use 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 has no methods optimized and thus its *IPC* metric would be similar to the run with the baseline compiler; and (ii) at the end of the run the AOS has optimized all the hot methods and thus its *IPC* metric would 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 that used the baseline and optimizing JIT compilers did not exhibit the gradual increase pattern, indicating that the AOS is the cause of the pattern.

4.4 Discussion

There are a number of lessons that can be learnt from our example. First, trace-information was essential to identify the performance anomaly. Using only aggregate-information could never have identified the performance anomaly because the *IPC* changed gradually over time. Second, correlating trace-information with the performance anomaly was essential to both disprove hypotheses (e.g., transactional behavior was not the cause) and to quickly focus on a few potential candidates from a large set of signals (e.g. identify the *LsuFlush/Cyc*). Even though there are hundreds of POWER4 hardware performance monitor events, correlation helped us to quickly identify the *LsuFlush/Cyc* metric as a potential candidate for causing the performance anomaly. Third, performance analysis is an iterative process. The vertical profiling process had to be reapplied when a new causal metric was found, until the performance anomaly could be fully explained. In our example, two iterations were required to find the two causal metrics: *LsuFlush/Cyc* and *UnoptMonitorEnter*. Fourth, multiple layers of the execution stack may need to be examined to understand a performance anomaly. Looking only at the metrics in one layer of the execution stack does not provide the complete picture of performance. Finally, each layer of the execution stack and even different components in a layer require expert knowledge. Understanding the meaning of the *LsuFlush/Cyc* metric was essential to determine where to look in the software stack to find the cause of the performance anomaly. Understanding how the adaptive optimization system works was essential to determining what metrics to use in JVM layer.

5. RELATED WORK

Programmers have been using general purpose profiling tools for at least three decades to understand and improve the performance of their applications. In this section, we argue that information provided by these prior tools is inadequate for understanding the performance of commercial applications. Specifically, to understand the performance of commercial applications, the information that is collected about the application's execution must satisfy the following requirements: (i) not lose relevant information, and (ii) distinguish between the different layers of the execution stack (hardware and software), and their components.

5.1 Inadequacy of Aggregate-Information

Aggregate-information for a particular metric is a single value that aggregates (e.g. averages or counts) the behavior of that metrics throughout the execution of an application. Profilers, such as the commonly-used *grpof* [14], compute aggregate information by reporting the time spent in each routine (or caller-callee pair). The time spent in a routine is aggregated because it combines information from every call to that routine into a single value.

Unfortunately, aggregate-information may not satisfy either of the above two requirements: (i) aggregate-information loses relevant information by not capturing any change in behavior over time; (ii) aggregate-information is usually collected from only some of the relevant subsystems (e.g., *grpof* just computes information from the application level).

5.2 Inadequacy of Trace-Information

Trace-information captures a metric's value in discrete intervals over time. At one extreme, trace data can capture a metric's value every time an event occurs, such as every time an instruction executes or a memory address is read. At the other extreme, trace-information for a metric can be the count of the number of times an event occurs for a given time interval, for example, every ten milliseconds. Aggregate-information can be thought of as trace data where the time interval is the length of the application's execution.[4] There are a number of tools that collect trace-information. For example, qpt [4], eel [13], atom [8], or shade [6] capture instruction and address traces. In addition, JVMPI [1] is Java virtual machine based that captures language level events such as method enter and exit, and object allocation. Because trace-information captures a metric's value over time, requirement (i) is satisfied as long as the time interval between which a metric's values are recorded is small enough.

In the past, trace-information has not satisfied requirement (ii). Typically, trace-information for different layers of the execution stack are collected independently and therefore can not be correlated because it is not clear what element in the trace of one layer can be correlated with an element in the trace of another layer of the execution stack. For example, existing general purpose tracing tools such as qpt, eel, atom, or shade generate trace-information from only some layers of the execution stack; for example, atom is invaluable for generating traces from the application layer. However, these tools typically do not include events from the operating system (e.g. page faults) or from the hardware subsystems (e.g., flush of the icache). Thus, using these tools the programmer gets an incomplete picture of their application's behavior impact on the underlying hardware. In our experience, trace-information from a subset of the layers in the execution stack is inadequate for understanding the performance of commercial applications.

6. CONCLUSIONS

This paper used a performance anomaly as an example to illustrate how one approach, vertical profiling, was used to determine the cause of the performance anomaly. Our example demonstrates that using commercial applications to evaluate and understand computer architectures is a difficult task. Determining the cause required looking at multiple layers of the execution stack (hardware, JVM and application). In each layer, expert knowledge is required to be able to determine if a metric in that layer is the cause of a performance anomaly.

[4]There is actually another dimension, sampling, that does not require exhaustive trace-information; that is, the occurrence of every event of a metric may not be accounted for in the sampled trace-information. Sampling, and all of its variants, takes advantage of the 90-10 rule that 90% of the time the application is doing the same thing, and therefore the probability that sampling will identify what is being done the majority of the time is high.

Our example, demonstrated why both the identification and the solution of the performance anomaly required trace-information, because the analysis process needed to reason about behavior as it changes over time. Aggregate-information alone would not suffice. Even with trace-information, techniques are needed to determine which metric is causing the performance anomaly. Vertical profiling proposes using correlation to identify potential candidates for the cause. Currently a human being needs to inspect the set of candidates to determine which are causal.

The performance anomaly required multiple iterations of the vertical profiling process. Initially, the *LsuFlush/Cyc* metric was identified as causing the performance anomaly at the hardware level, but did not identify what was causing the performance anomaly at the software level. Nevertheless, *LsuFlush/Cyc* did give us insight into where to look in the software stack. Our initial focus, the application layer, turned out to be incorrect; however, this was quickly validated with correlation. Our second attempt found the cause of the performance anomaly in the adaptive optimization system of the Java virtual machine.

Evaluating computer architectures is difficult because the complexity of the software stack and hardware is continuing to increase. Commercial applications are increasingly using software developed by third parties and the impact of the interaction between the applications and third party software on the behavior of computer architectures is not always readily known. To boost performance, current microprocessors are adding simultaneous multithreading and multiple cores per chip without knowing how the interaction of multiple applications that run on the same processor impact performance due to resource sharing. This example has demonstrated that new techniques, such as vertical profiling, have to be developed to be able to evaluate modern computer architectures.

7. FUTURE WORK

There is no question that sophisticated tools are needed to understand today's complex software and hardware systems. Even with the correct tools, evaluating computer architectures using commercial applications is hard, because expert knowledge is required in may domains and a tremendous amount of manual effort is required to track down performance anomalies. Our goal is to automate performance analysis by capturing expert knowledge so that it can be automatically applied to understand performance anomalies and to automate the manual steps that are performed by performance analysts. We have commenced this work by starting to automate vertical profiling [10]. Specifically, we are developing techniques to automatically aligning traces and to automatically or semi-automatically correlate metrics.

8. ACKNOWLEDGEMENTS

The authors would like to thank Michael Hind for his feedback on this paper. This work was supported by Defense Advanced Research Project Agency Contract NBCH30390004, NSF CSE-0509521, NSF Career CCR-0133457, and a gift from Intel. 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] Java virtual machine profiler interface (jvmpi). http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.

[2] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. 35(10):47–65, October 2000. Proceedings of the 2000 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'00).

[3] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 92(2):449–466, Feb. 2005.

[4] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(4):1319–1360, 1994.

[5] Călin Caşcaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the Fourteenth International Conference on Parallel Architectures and Compiler Technology (PACT05)*. IEEE, Sep. 2005.

[6] Robert Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *ACM SIGMETRICS Performance Evaluation Review*, 22(1):128–137, May 1994.

[7] Standard Performance Evaluation Corporation. SPECjbb2000 (Java Business Benchmark). http://www.spec.org/jbb2000.

[8] Alan Eustace and Amitabh Srivastava. Atom: A flexible interface for building high performance program analysis tools. In *USENIX Winter*, pages 303–314, 1995.

[9] Matthias Hauswirth. *Understanding Program Performance Using Temporal Vertical Profiling*. PhD thesis, University of Colorado at Boulder, 2005.

[10] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA05)*. ACM Press, October 2005.

[11] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA04)*. ACM Press, October 2004.

[12] Jikes Research Virtual Machine (RVM). http://www.ibm.com/developerworks/oss/jikesrvm.

[13] James R. Larus and Eric Schnarr. Eel: Machine-independent executable editting. In *Proceedings of the SIGPLAN Conference on Programming Languages Design and Implementation (PLDI95)*. ACM Press, June 1995.

[14] P. Kessler S. Graham and M. McKusick. Gprof: A call graph execution profiler. *Proceeedings of the SIGPLAN '82 Symposium on Compiler Construction*, 17(6):120–126, June 1982.

[15] 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.