Using Hardware Performance Monitors to Understand the Behavior of Java Applications

Peter F. Sweeney[1] Matthias Hauswirth[2]*
Brendon Cahoon[1] Perry Cheng[1] Amer Diwan[2] David Grove[1] Michael Hind[1]
[1]*IBM Thomas J. Watson Research Center*
[2]*University of Colorado at Boulder*

Abstract

Modern Java programs, such as middleware and application servers, include many complex software components. Improving the performance of these Java applications requires a better understanding of the interactions between the application, virtual machine, operating system, and architecture. Hardware performance monitors, which are available on most modern processors, provide facilities to obtain detailed performance measurements of long-running applications in real time. However, interpreting the data collected using hardware performance monitors is difficult because of the low-level nature of the data.

We have developed a system, consisting of two components, to alleviate the difficulty of interpreting results obtained using hardware performance monitors. The first component is an enhanced VM that generates traces of hardware performance monitor values while executing Java programs. This enhanced VM generates a separate trace for each Java thread and CPU combination and thus provides accurate results in a multithreaded and multiprocessor environment. The second component is a tool that allows users to interactively explore the traces using a graphical interface. We implemented our tools in the context of Jikes RVM, an open source Java VM, and evaluated it on a POWER4 multiprocessor. We demonstrate that our system is effective in uncovering as yet unknown performance characteristics and is a first step in exploring the reasons behind observed behavior of a Java program.

1 Introduction

Modern microprocessors provide hardware performance monitors (HPMs) to help programmers understand the low-level behavior of their applications. By counting the occurrences of events, such as pipeline stalls or cache misses, HPMs provide information that would otherwise require detailed and, therefore, slow simulations. Because the information provided by HPMs is low-level in nature, programmers still have the difficult task of determining how the information relates to their program at the source level. This paper describes a system that alleviates some of this difficulty by relating HPM data to Java threads in a symmetric multiprocessor environment (SMP).

Our system overcomes the following four challenges in interpreting HPM data for Java programs. First, because a Java virtual machine's rich runtime support uses the same hardware resources as the application, resource usage of the VM threads needs to be distinguished from those of the application. Second, because Java applications often employ multiple threads, each thread's resource usage needs to be distinguished. Third, because the characteristics of even a single Java thread may vary during its execution it is important to capture the time-varying behavior of a thread. Fourth, because in a SMP environment a single Java thread may migrate among several physical processors, the performance characteristics of each thread and CPU combination need to be attributed correctly.

Our system consists of two components: an enhanced VM that generates traces of hardware events and a visualization tool that processes these traces. The enhanced VM, an extension to Jikes RVM (Research Virtual Machine), accesses the PowerPC HPMs to generate a trace from a running application. The trace consists of a sequence of trace records that capture hardware performance events during the length of a thread scheduler quantum for each processor. In addition to calculating aggregate metrics, such as overall IPC (instruction per cycle) for each thread, these traces allow one to explore the time-varying behavior of threads, both at application and VM level.

The output of the trace generator is too large to be immediately usable. For example, one thirty-second run has almost sixty thousand events in its trace. The visualization tool allows users to interactively explore the traces and to compare multiple metrics (e.g., cache misses and memory stalls) graphically and side-by-side. In this way a user can explore hypotheses interactively (e.g., are metrics A and B correlated?).

We demonstrate the usefulness of the system by applying it

*Work done while a summer student at IBM Research in 2003.

to a variation of the SPECjbb2000benchmark. We show that the system is effective in identifying performance anomalies and also helps us to explore their cause.

To summarize, our contributions are as follows:

- We describe an extension to Jikes RVM to access PowerPC HPMs and accurately attribute them to Java threads in a SMP environment. Although many prior systems (such as DCPI [5] and OProfile [26]) give users access to HPMs, we believe ours is the first system that generates Java thread-specific traces over time of HPM data for multithreaded applications on a multiprocessor.

- We present the `Performance Explorer`, a visualization tool for interactively and graphically understanding the data.

- We use our tools to identify performance anomalies in our benchmark and also to explore the cause of the anomalies. Explaining the cause of the anomalies was tricky due to the complexity of both the software (which uses hard to understand features such as adaptive compilation and garbage collection) and the hardware (which employs an elaborate memory system design). Our visualization tool was invaluable in this exploration, but we realize that there is still more work to be done in this area.

The rest of this paper is organized as follows. Section 2 provides the background for this work, including an overview of Jikes RVM and the existing mechanism for accessing the PowerPC HPMs under AIX. Section 3 describes the design and implementation of the VM extension mechanism for recording HPMs. Section 4 introduces the visualization tool. Section 5 illustrates how the tool can be used to help understand the hardware performance of Java applications. Section 6 discusses related work. Section 7 outlines avenues for future work and Section 8 draws some conclusions.

2 Background

This section provides the background for this work. Section 2.1 provides an overview of the existing Jikes RVM infrastructure that this paper uses as a foundation. Section 2.2 summarizes hardware performance monitors on AIX.

2.1 Jikes RVM

Jikes RVM [19] is an open source research virtual machine that provides a flexible testbed for prototyping virtual machine technology. It executes Java bytecodes and runs on the Linux/IA32, AIX/PowerPC, Linux/PowerPC platforms, and OS X/PowerPC platforms. This section briefly provides background on a few relevant aspects of Jikes RVM. More details are available at the project web site [19] and in survey papers [2, 12, 6].

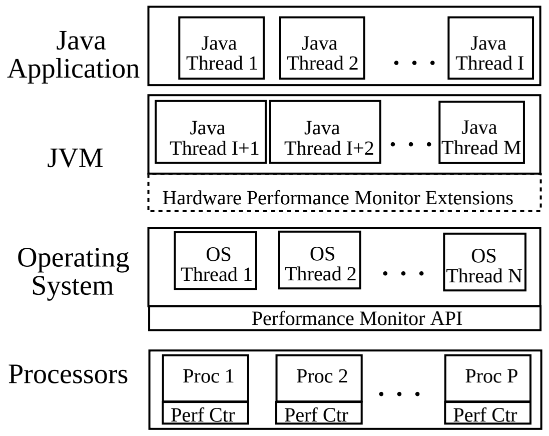


Figure 1: Architectural overview.

Jikes RVM is implemented in the Java programming language [3] and uses Java threads to implement several subsystems, such as the garbage collector [10] and adaptive optimization system [6]. Thus, our HPM infrastructure provides insight both into the performance characteristics of Java programs that execute on top of Jikes RVM and into the inner workings of the virtual machine itself. In particular, by gathering per-Java-thread HPM information the behavior of these VM threads is separated from application threads. However, VM services, such as memory allocation, that execute as part of the application thread are not separately distinguished.

As shown in Figure 1, 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. A command line argument to Jikes RVM specifies the number of Pthreads, and corresponding kernel threads, that Jikes RVM creates. 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 processors* 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. Thus, to force one of the currently executing Java threads to stop running, the system sets this flag and waits for the thread to execute a yieldpoint sequence. 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. Because yieldpoints are a subset of the GC safe points, i.e., program points where the stack references (or GC roots) are known, only the running Java threads need to reach a GC safe point to begin a stop-the-world garbage collection; threads in the scheduler's ready queue are already prepared for a garbage collection.

2.2 AIX Hardware Performance Monitors

Most implementations of modern architectures (e.g. PowerPC POWER4, IA-64 Itanium) provide facilities to count hardware events. Examples of typical events that may be counted include processor cycles, instructions completed, and L1 cache misses. An architecture's implementation exposes a software interface to the event counters through a set of special purpose hardware registers. The software interface enables a programmer to monitor the performance of an application at the architectural level.

The AIX 5.1 operating system provides a library with an application programming interface to the hardware counters as an operating system kernel extension (pmapi).[1] The API, shown as part of the operating system layer in Figure 1, provides a set of system calls to initialize, start, stop, and read the hardware counters. The initialization function enables the programmer to specify a list of predefined events to count. The number and list of events depends on the architecture's implementation, and vary substantially between different PowerPC implementations (e.g. PowerPC 604e, POWER3, and POWER4). The API provides an interface to count the events for a single kernel thread or for a group of threads. The library automatically handles hardware counter overflows and kernel thread context switches.

A programmer can count the number of times a hardware event occurs in a code segment by manually instrumenting a program with the appropriate API calls. Prior to the code segment, the instrumentation calls the API routines to initialize the library to count certain events and to commence counting. After the code segment, the instrumentation calls the API routines to stop counting, read the hardware counter events, and optionally print the values. The HPM toolkit provides a command line facility to measure the complete execution of an application [15].

Some processors provide more sophisticated facilities to access HPM data. These facilities include thresholding and sampling mechanisms. The thresholding mechanism allows the programmer to specify a threshold value and an event. Only if the event exceeds the threshold value is the hardware counter incremented. The sampling mechanism allows the programmer to specify a value, n, and an event. When the event occurs for the nth time, the hardware exposes the executing instruction and operand address to the software. The POWER4 architecture provides thresholding and sampling capabilities, but the AIX 5.1 kernel extension library (pmapi) does not support them.

[1]The package is also available for earlier versions of AIX at IBM's AlphaWorks site `www.alphaworks.ibm.com/tech/pmapi`.

3 VM Extensions

This section describes extensions to Jikes RVM that enable the collection of trace files containing thread-specific, temporally fine-grained HPM data in an SMP environment. The section begins by enumerating the design goals for the infrastructure. It then describes the trace record format and highlights some of the key ideas of the implementation. Finally, it considers issues that may arise when attempting to implement similar functionality in other virtual machines.

3.1 Design Goals

There are four primary goals for the HPM tracing infrastructure.

Thread-specific data The infrastructure must be able to discriminate between the various Java threads that make up the application. Many large Java applications are multi-threaded, with different threads being assigned different portions of the overall computation.

Fine-grained temporal information The performance characteristics of a thread may vary over time. The infrastructure must enable the identification of such changes.

SMP Support The infrastructure must work on SMPs. In an SMP environment, multiple threads will be executing concurrently on different physical processors, and the same Java thread may execute on different processors over time. There will be a stream of HPM data associated with each virtual processor and it must be possible to combine these separate streams into a single stream that accurately reflects the program's execution.

Low overhead Low overhead is desirable both to minimize the perturbations in the application introduced by gathering the data and to enable it to be used to gather traces in production environments or for long-running applications.

3.2 Trace Files

When the infrastructure is enabled, it generates a trace file for each Jikes RVM virtual processor, and one meta file. This section describes the structure of the trace and meta files, and discusses how the data gathered enables the system to meet the design goals enumerated in Section 3.1.

The core of the trace file is a series of trace records. Each trace record represents a measurement period in which exactly one Java thread was executing on the given virtual processor. The HPM counters are read at the beginning and end of the measurement period and the change in the counter values is reported in the trace record. A trace record contains the following data:

Virtual Processor ID This field contains the unique ID of the virtual processor that was executing during the measurement period. Although this information can be inferred (each trace file contains trace records from exactly one virtual processor), we chose to encode this in the trace record to simplify merging multiple trace files into a single file that represents an SMP execution trace.

Thread ID This field contains the unique ID of the thread that was executing during the measurement period.

Thread Yield Status This boolean field captures if the thread yielded before its scheduling quantum expired. It is implemented by negating the thread ID.

Top Method ID This field contains the unique ID of the top method on the runtime stack at the end of the measurement period, excluding the scheduler method taking the sample. Because the complete stack is available, this field could be extended to capture more methods as was done to guide profile-directed inlining [18].

Real Time This field contains the value of the PowerPC time base register at the beginning of the measurement period represented by this trace record. The time base register contains a 64-bit unsigned quantity that is incremented periodically (at an implementation-defined interval) [22].

Real Time Duration This field contains the duration of the measurement period as measured by the difference in the time base register between the start and end of the measurement period.

Hardware Counter Values These fields contain the change in each hardware counter value during the measurement period. The number of hardware counters varies among different implementations of the PowerPC architecture. In most anticipated uses, one of the counters will be counting cycles executed, but this is not required by the infrastructure.

As each trace record contains hardware counter values for a single Java thread on a single virtual processor, we are able to gather thread-specific HPM data. The key element for SMP support is the inclusion in the trace record of the real time values read from the PowerPC time base register. The primary use of the real time value is to merge together multiple trace files by sorting the trace records in the combined trace by the real time values to create a single trace that accurately models concurrent events on multiple processors. A secondary use of this data is to detect that the OS has scheduled other Pthreads on the processor during the measurement interval. Large discrepancies between the real time delta and the executed cycles as reported by the hardware counter indicate that some OS scheduling activity has occurred and that the Pthread shared the processor during the measurement period. Recall that the OS extension already distinguishes counters for each OS kernel thread.

In addition to trace records containing hardware counter values, a trace file may also contain marker records. We provide a mechanism, via calls to the VM, for a Java thread to specify program points that when executed will place a marker record in the trace file. These marker trace records, which can contain an arbitrary string, allow the programmer to focus on particular portions of an execution's trace. Because this mechanism is available at the Java level, it can be used to filter both application and VM activities, such as the various stages of garbage collection.

A meta file is generated in conjunction with a benchmark's trace files. The meta file specifies the number of HPM counter values in each trace record and provides the following mappings: from counter to event name, from thread ID to thread name, and from method ID to method signature. The number of counters and the mappings provide a mechanism to interpret a trace record, reducing a trace record's size by eliminating the need to name trace record items in each individual trace record.

3.3 Implementation Details

To enable Jikes RVM to access the C pmapi API we defined a Java class with a set of native methods that mirrors the functionality of the pmapi interface, represented by the dashed box of the JVM in Figure 1. In addition to enabling our VM extensions in Jikes RVM to access these functions, the interface class can also be used to manually instrument arbitrary Java applications to gather aggregate HPM data. We are using this facility to compare the performance characteristics of Java applications when run on Jikes RVM and on other JVMs.

The main extension point in Jikes RVM was to add code in the thread scheduler's context switching sequence to read the hardware counters and real time clock on every context switch in the VM. This information is accumulated into summaries for each virtual processor and Java thread, and written into a per virtual processor trace record buffer. Each virtual processor has two dedicated 4K trace record buffers and a dedicated Java thread, called a *trace writer* thread, whose function is to transfer the contents of a full buffer to a file. Trace records for a virtual processor are written into an active buffer. When the buffer is full, the virtual processor signals the trace write thread and starts writing to the other buffer.

By alternating between two buffers, we continuously gather trace records with low overhead. By having a dedicated Java thread drain a full buffer, and thus, not suspend the current thread to perform this task, we avoid directly perturbing the behavior of the other threads in the system. This implementation also enables easy measurement of the overhead of writing the trace file because HPM data is gathered for all Java threads, including the threads that are writing the trace files. In our experiments 1.7% of all cycles are spent executing the trace writer threads. The overhead of reading the hardware counters and real time clock on every thread switch and storing the trace information into the buffer is in the measurement noise. Thus, the total overhead of the infrastructure is less than 2%.

Minor changes were also made in the boot sequence of the virtual machine, and in the code that creates virtual processors and Java threads to appropriately initialize data structures. The VM extensions have been available in Jikes RVM [19] since the 2.2.2 release (June 2003).

The disk space required to store trace records is a function of the trace record size, the frequency of thread switches, and the number of virtual processors. For example, running one warehouse in the SPECjbb2000 benchmark on one virtual processor, we found that 12 Kbytes per second were written with a 10 millisecond scheduling quantum.

3.4 Discussion

Jikes RVM's M-to-N threading required an extension of the virtual machine to gather Java thread specific HPM data. In JVMs that directly map Java threads to Pthreads, it should be possible to gather *aggregate* Java thread specific HPM data using the pmapi library by making relatively simple extensions to read HPM counters when threads are created and terminated. So, in this respect the Jikes RVM implementation was more complex than it might have been in other JVMs. However, M-to-N threading made the gathering of fine-grained temporal HPM data fairly straightforward. A relatively simple extension to the context-switching sequence to read the HPM counters on every thread switch was sufficient to collect the desired data. Gathering this kind of data on virtual machines that do not employ M-to-N threading will probably be significantly more difficult because applying a similar design would require modifications to either the Pthread or OS thread libraries.

4 Visualization Tool

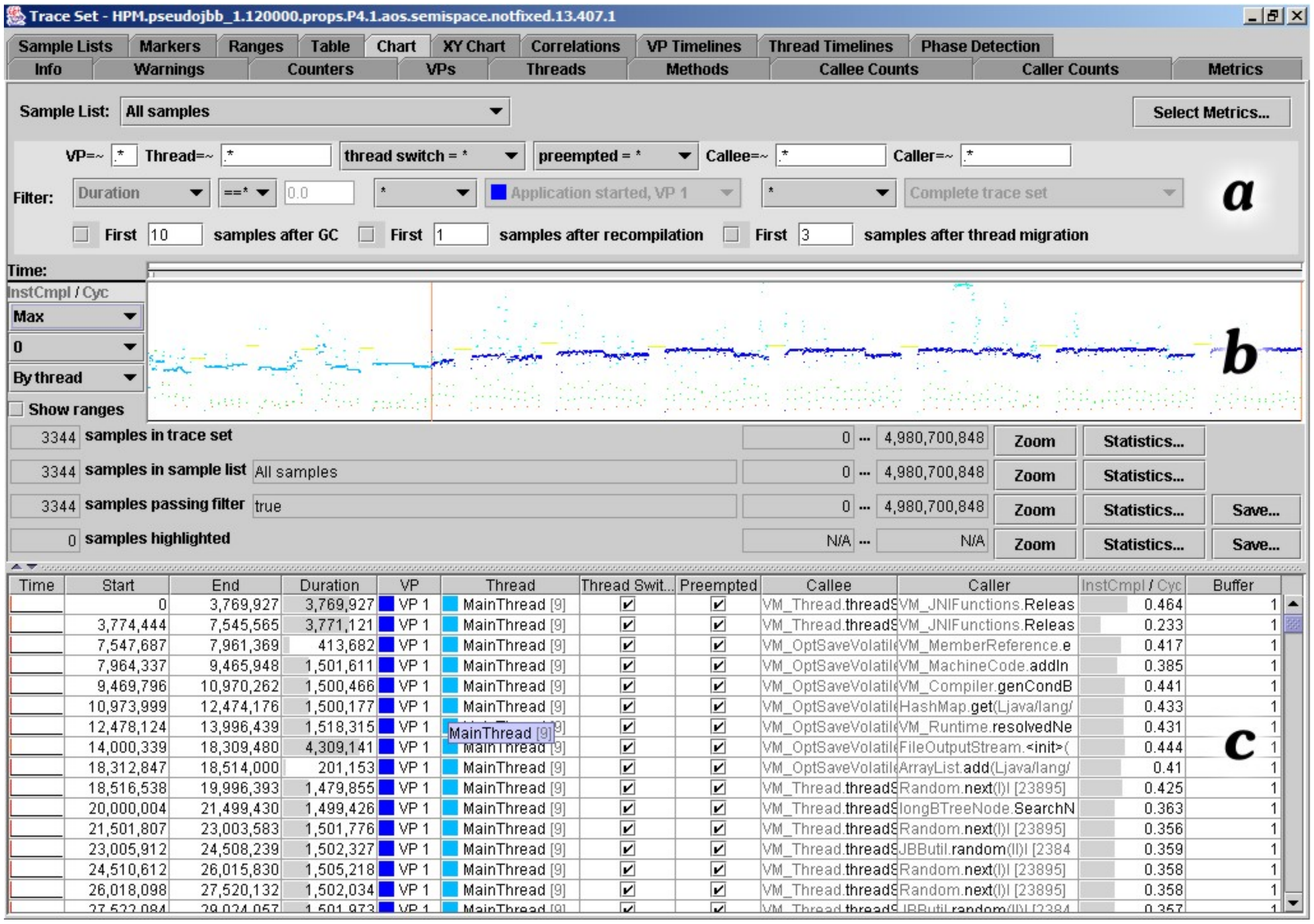
This section describes our visualization tool, called the `Performance Explorer`, which supports performance analysis through the interactive visualization of trace records that are generated by the VM extensions described in Section 3. Figure 2 presents an overview of the `Performance Explorer` when run with a variant of SPECjbb2000 using one warehouse on one virtual processor. The figure comprises three parts (`a`, `b`, and `c`), which are further described below.

The `Performance Explorer` is based on two key concepts: *trace record sets* and *metrics*. A *trace record set* is a set of trace records. Given the trace files of an application's execution, the `Performance Explorer` provides an initial trace record set containing all trace records of all threads on all virtual processors. It also provides filters to create subsets of a trace record set (e.g., all trace records of the MainThread, all trace records longer than 5 ms, all trace records with more than 1000 L1 D-cache misses, or all trace records ending in a Java method matching the regular expression "`.*lock.*`"). Part `a` of Figure 2 illustrates the user interface for configuring these filters. Furthermore, the `Performance Explorer` provides set operations (union, intersection, difference) on trace record sets.

A *metric* extracts or computes a value from a trace record. The `Performance Explorer` provides a metric for each hardware counter gathered in the trace files, and a metric for the trace record duration. The user can define new metrics using arithmetic operations on existing metrics. For example, instructions per cycle (IPC) can be computed using a computed metric that divides the instructions completed (INST_CMPL) event value by the cycles (CYC) event value. Part `b` in Figure 2 shows a graph for just one metric. The horizontal axis is wall-clock time, and the vertical axis is defined by the metric, which in this case is IPC. The vertical line that is a quarter of the way in from the left side of the graph represents a marker trace record generated by manual instrumentation of the program. This specific marker shows when the warehouse application thread starts executing. To the right of the marker, the applications enters a steady state where the warehouse thread is created and executed. To the left of the marker represents the program's start up where its main thread dominates execution. Each line segment in the graph (in this zoomed-out view of the graph most line segments appear as points) represents a trace record. The length of the line segment represents its wall clock duration. The color of the line segment (different shades of gray in this paper) indicates the corresponding Java thread. The user can zoom in and out and scroll the graph.

Part `c` of Figure 2 displays a table of the trace records visualized in the graph, one trace record per line. This table presents all the attributes of a trace record, including the values of all metrics plotted in the graph. Selecting a range of trace records in the graph selects the same records in the table, and vice versa. This allows the user to select anomalous patterns in the graph, for example the drop in IPC before each garbage collection, and immediately see all the attributes (like method names) of the corresponding trace records in the trace record table. The user can get simple descriptive statistics (sum, minimum, maximum, average, standard deviation, and mean delta) over the selected trace records for all metrics. Finally, a selection of trace records can be named and saved as a new trace record set.

In addition to providing time graphs and trace record tables, the `Performance Explorer` also provides several other ways to visualize the trace data. The `Performance Explorer` provides thread timelines, which are tables where each column represents a thread, each row represents an interval in time, and the cells are colored based on the processor that is executing the thread. The `Performance Explorer` provides processor timelines, where each column represents a processor, and the cells are colored based on the thread that executes on the processor. Cells in these timelines visualize a given metric (like IPC), and they are adorned with glyphs to show preemption or yielding of a thread at the



Sample List: All samples

VP=~ .* Thread=~ .* thread switch = * preempted = * Callee=~ .* Caller=~ .*

Filter: Duration ==* 0.0 * Application started, VP 1 * Complete trace set

First 10 samples after GC First 1 samples after recompilation First 3 samples after thread migration

Time:

InstCmpl / Cyc: Max, 0, By thread, Show ranges

3344	samples in trace set		0	4,980,700,848
3344	samples in sample list	All samples	0	4,980,700,848
3344	samples passing filter	true	0	4,980,700,848
0	samples highlighted		N/A	N/A

Time	Start	End	Duration	VP	Thread	Thread Swit...	Preempted	Callee	Caller	InstCmpl / Cyc	Buffer
	0	3,769,927	3,769,927	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	VM_JNIFunctions.Releas	0.464	1
	3,774,444	7,545,565	3,771,121	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	VM_JNIFunctions.Releas	0.233	1
	7,547,687	7,961,369	413,682	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	VM_MemberReference.e	0.417	1
	7,964,337	9,465,948	1,501,611	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	VM_MachineCode.addIn	0.385	1
	9,469,796	10,970,262	1,500,466	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	VM_Compiler.genCondB	0.441	1
	10,973,999	12,474,176	1,500,177	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	HashMap.get(Ljava/lang/	0.433	1
	12,478,124	13,996,439	1,518,315	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	VM_Runtime.resolvedNe	0.431	1
	14,000,339	18,309,480	4,309,141	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	FileOutputStream.<init>(	0.444	1
	18,312,847	18,514,000	201,153	VP 1	MainThread [9]	✓	✓	VM_OptSaveVolatile	ArrayList.add(Ljava/lang/	0.41	1
	18,516,538	19,996,393	1,479,855	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.425	1
	20,000,004	21,499,430	1,499,426	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	longBTreeNode.SearchN	0.363	1
	21,501,807	23,003,583	1,501,776	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.356	1
	23,005,912	24,508,239	1,502,327	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	JBButil.random(II)I [2384	0.359	1
	24,510,612	26,015,830	1,505,218	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.358	1
	26,018,098	27,520,132	1,502,034	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	Random.next(I)I [23895]	0.358	1
	27,522,084	29,024,057	1,501,973	VP 1	MainThread [9]	✓	✓	VM_Thread.threadS	JBButil.random(II)I [2384	0.357	1

Figure 2: Overview of the `Performance Explorer`.

end of a time slice. These timelines are helpful in analyzing scheduling effects. The `Performance Explorer` provides scatter plots, where the X and Y axes can be any given metric, and all trace records of a trace record set are represented as points. Lastly, the `Performance Explorer` calculates the correlations between any two metrics over a trace record set.

Due to the extensive number of HPM events that can be counted on POWER4 processors, and the limitation of a given event being available only in a limited number of hardware counters, the `Performance Explorer` provides functionality for exploring the available events and event groups.

Because of space considerations, subsequent figures only contain information from the `Performance Explorer` that is pertinent to the discussion at hand.

5 Experiments

This section demonstrates how we used the `Performance Explorer` to understand the performance behavior of a variant of the SPECjbb2000 benchmark on a PowerPC POWER4 machine.

Caches	Size	Line size	Kind
L1 Instr.	32KB	128B	direct mapped
L1 Data	64KB	128B	2-way set assoc.
L2 Unified	1.5MB	128B	8-way set assoc.
L3 local	32MB	512B	8-way set assoc.
L3 remote	32MB	512B	8-way set assoc.

Table 1: POWER4 three levels of cache hierarchy.

5.1 POWER4

The POWER4 [8] is a 64-bit microprocessor that contains two processors on each chip. Four chips can be arranged on a module to form an 8-way machine, and four modules can be combined to form a 32-way machine. The POWER4 contains three cache levels as described in Table 1. There is one L1 data and one L1 instruction cache for each core on a chip. The L1 caches are *store through* (write through); that is, a write to the L1 is stored through to L2. Two processors on a chip share an L2 cache. The L2 cache is inclusive of the two L1 data caches, but not inclusive of the two L1 instruction caches; that is, any data in the L1 data cache also resides in the L2 cache. However, data may reside in the L1 instruction cache that does not reside in the L2 cache. The L2 cache is *store in* (write back); that is, a write to L2 is not written to main memory (or L3). Up to four L3 caches can be arranged on a module. The L3 cache acts as a victim cache, storing cache lines that are evicted from L2; therefore, the L3 cache is not inclusive of the L2 cache.

5.2 Experimental Methodology

We used a 4-way POWER4 machine with two chips in which each chip is placed on a separate module. We use a variant of the SPECjbb2000 [29] benchmark for our experiments. SPECjbb2000 simulates a wholesale company whose execution consists of two stages. During startup, the main thread sets up and starts a number of warehouse threads. During steady state, the warehouse threads execute transactions against a database (represented as in-memory binary trees). The variant of SPECjbb2000 that we use is called pseudojbb: pseudojbb runs for a fixed number of transactions (120,000) instead of a fixed amount of time. We run pseudojbb with one warehouse thread on a single virtual processor. In our configuration, the pseudojbb run takes 25 seconds on a POWER4 workstation. We use an adaptive Jikes RVM configuration that has the adaptive optimization system (AOS) with a semispace garbage collector. The source code for Jikes RVM is from the September 26, 2003 head of the public CVS repository.

When run on a single virtual processor, Jikes RVM creates eight Java threads during execution in addition to the two Java threads created by pseudojbb: a *garbage collection* thread that executes only during garbage collection; a *finalizer* thread that finalizes dead objects and is executed infrequently; a *debugger* thread that never executes; and an *idle* thread that helps load balance Java threads when a virtual processor is idle, but because only a single virtual processor is used the idle thread never executes. As mentioned in Section 3 the VM also creates a *trace writer* thread to transfer trace records from a buffer to a trace file. The last three threads are related to the adaptive optimization system [6]. The *compilation* thread performs optimized compilations of methods selected by the *controller* thread, which processes online profile data recorded by the *organizer* thread.

5.3 Exploration

This section demonstrates two approaches to using the `Performance Explorer` to investigate performance phenomena. The first approach uses the `Performance Explorer` to look for general trends over time. We discovered the following two unexpected instruction per cycle (IPC) trends:

- Significant IPC improvement over time for an adaptive Jikes RVM configuration.
- Significant IPC degradation before GC.

Both trends were unknown before using the `Performance Explorer`. These trends over time would be difficult, if not impossible, to detect without a visualization tool.

The second approach uses the `Performance Explorer` to explore memory latency issues with respect to the POWER4 microarchitecture. Specifically, we used the `Performance Explorer` to explore the following questions:

- What is the impact on IPC and memory performance of context switches between different Java threads?[2]
- How do latency issues manifest in the POWER4 memory hierarchy?

Although the answers to these two questions might have been determined by performing a computation over the trace records directly, the `Performance Explorer` provides a unified approach to explore these questions.

5.4 General Performance Trends

The top graph in Figure 3 presents a graph of the IPC of a warehouse thread over time. For clarity of presentation, only the IPC trace records of the warehouse thread are displayed; all other Java threads are not shown. The noticeable gaps in the warehouse trace records are stop-the-world garbage collections, which disable Java thread scheduling when it runs.

Two general performance anomalies are noticeable in this graph. First, IPC improves over time. The IPC in the left corner is around 0.41, while the IPC in the right corner is above 0.49, a 20% increase. Second, before each of the GCs there is a significant IPC degradation or drop. We used the `Performance Explorer` to help us explore both of these phenomena.

5.4.1 Anomaly 1

To understand the IPC improvement over time, we used the `Performance Explorer` to chart each HPM event to determine the event or events that have a high correlation with the IPC. We found two such HPM events. The bottom graph in Figure 3 presents a warehouse thread's flushes per instructions completed over time.

A flush event may occur in a out-of-order processor for multiple reasons. A common reason is an out-of-order memory operation that executes and violates a data dependence. For example, the load of a memory location will be flushed if the load is speculatively executed before the execution of the store instruction to that memory location. When an instruction is flushed, all the instructions that are dispatched and occur in program order before the flushed instruction are also flushed. Hence, a flush event is expensive.

[2]In the past, a context switch, when control changes from one process to another, had been identified as having an impact on performance due to destroyed cache locality [25].

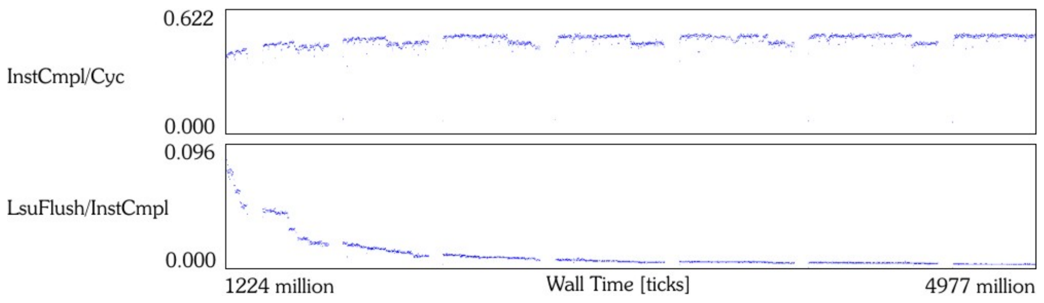


Figure 3: A graph over time of the warehouse thread's instruction per cycle and flushes per instructions completed.

Metric	JIT-base	AOS start (Δ%)	AOS end (Δ%)	JIT-opt0 (Δ%)
IPC	0.348	0.415 (+19.3%)	0.489 (+42.5%)	0.500 (+43.7%)
LSU_FLUSH/INST_CMPL	15.2%	7.0% (-54.2%)	0.3% (-97.4%)	0.1% (-99.3%)
INST_CMPL	4.18M	4.05M (+16.0%)	5.84M (+42.6%)	5.96M (+42.6%)

Table 2: Metrics across JIT and AOS configurations.

The POWER4 has multiple HPM events that count flush events. We have added together the values of the two dominant HPM flush events so that only one graph is displayed. The bottom graphs in Figure 3 illustrate that the flush rate changes dramatically over time. In the left corner, a flush event occurs for as many as 9.6% of the completed instructions while in the right corner a flush event occurs for as little as 0.4% of the completed instructions.

The Jikes RVM's adaptive optimization system (AOS) [6] compiles a method baseline (non-optimizing) compiler when it is first invoked. At thread switch time, the system records the method ID of the top method on the call stack and uses the number of times a method is sampled to predict how long a method will execute in the future. Using a cost/benefit model, the system determines when compiling a method at a particular optimization level has a benefit that outweighs the cost of the compilation for the expected future execution of the method. The system samples method IDs continuously throughout an application's execution, and optimizes methods whenever the model deems the optimizations are beneficial.

Table 2 provides insight into why the warehouse thread's performance improves over time. The first column specifies the two metrics that are graphed in Figure 3 and an additional metric of instructions completed (INST_CMPL). The table has four columns that contain metrics for the start and end of a run using the adaptive optimization system and two separate runs that use a non-adaptive strategy, which we call JIT configurations. In a JIT configuration, a method is compiled only once when it is first invoked. A JIT-base configuration uses the baseline compiler. A JIT-opt0 configuration uses the optimizing compiler at optimization level 0.[3] On a POWER4, the average execution time for SPECjbb2000 when one warehouse is run on 120,000 transactions is 27 seconds with JIT-opt0, and 175 seconds with JIT-base.

For this table, the metrics are computed as the average across warehouse trace records. The AOS start and end metrics were computed by taking the average of the first 14 and the last 14 warehouse trace records, respectively. The metrics for each of the JIT configurations is computed as the average over all of its warehouse trace records.

The first observation is that there is a large difference between the execution behavior of baseline compiled code (JIT-base) and the optimization level 0 compiled code (JIT-opt0). In particular, there is a 43.7% increase in IPC, a 99.3% decrease in flush events, and a 42.6% increase in instructions completed when going from baseline to optimization level 0. To understand this difference, we need to know how baseline compiled code differs from optimized code. The baseline compiler directly translates byte codes into machine code without any optimizations. In particular, no register allocation is performed and the Java expression stack is explicitly maintained with loads and stores accessing memory. Typically, after a value is pushed onto the stack it is popped off immediately and used. With baseline-compiled code, the number of flush events is high, 15.2% of the instructions completed. This is because the scheduling of out-of-order memory operations does not take into account the dependencies between memory locations: that is, the load instruction that models a pop to stack location $\mathcal{L}$ may be speculatively executed before the store instruction that models the push to stack location $\mathcal{L}$. In optimized code, the number of flush

[3]For the execution of pseudojbb that uses the adaptive optimization system, 612 methods are baseline compiled, and of those, 251 methods are recompiled: 125 methods are compiled at optimization level 0, 109 at level 1, and 17 at level 2. We show the metrics for a JIT-opt0 because the metrics at JIT-opt1 are similar, and although the metrics for JIT-opt2 degrade slightly, few methods are optimized at optimization level 2.

Metric	Total	Drop	Δ
INST_CMPL/CYC (IPC)	0.4924	0.4610	-6.4%
HV_CYC/CYC	0.0239	0.1249	+423.0%
EE_OFF/CYC	0.0197	0.0785	+300.0%
GRP_DISP_BLK_SB_CYC/CYC	0.0060	0.0258	+333.0%
LSU_SRQ_SYNC_CYC/CYC	0.0061	0.0170	+178.0%
STCX_FAIL/STCX_PASS+STCX_FAIL	0.0009	0.0040	+362.0%
LSU_LRQ_FULL_CYC/CYC	0.0008	0.0027	+250.0%

Table 3: Metrics that impact performance degradation before GC.

events is almost zero because values are loaded from memory into registers without explicitly modeling the Java expression stack.

Comparing the JIT configuration metrics, the impact of flush events on the number of completed instructions in a full 10 millisecond time slice is enormous:[4] 42.6% more instructions complete, when flushes are almost completely eliminated. In the AOS configuration, the execution behavior of the last fourteen warehouse trace records is very similar to the behavior of JIT-opt0. This is what is to be expected as the adaptive optimization system has had time to optimize the code that executes frequently. Running under the adaptive optimization system, when the warehouse thread starts executing all the warehouse methods are initially baseline compiled; however, because start up and steady state share code, some code that executes at the start of steady state is already optimized. As illustrated in Table 2, the AOS start has an expected behavior that falls between JIT-base and JIT-opt0 behavior.

5.4.2 Anomaly 2

To identify which HPM events correlate with the drop in IPC before each garbage collection, we used the `Performance Explorer` to select for each group of HPM events a representative set of warehouse trace records between two garbage collections and to compute the set's average for all HPM events (Total).[5] From this representative set, we used the `Performance Explorer` to pick the subset of trace records that represented the IPC drop (Drop) and computed its average for all HPM events. After these two computations, we identified the HPM events whose average values differed the most between Total and Drop. Table 3 presents our results. The first column identifies the HPM events. The HPM event values are normalized by dividing by cycles, except for STCX_FAIL, which is normalized by the number of STCX attempts. The second column presents the values for all trace records in the set (Total), the third column presents the values for the subset of trace records after IPC drops (Drop), and the final column presents how Drop's average changes as a percentage of Total's average. As can be seen, the IPC degraded by 6.4%, while there is a substantial increase (178–423%) in the percentage of the identified HPM events.

The set of events in Table 3 is eclectic. The HV_CYC (the processor is executing in hypervisor mode) and EE_OFF (the MSR EE bit is off) events specify the cycles spent in the kernel. The GRP_DISP_BLK_SB_CYC (dispatch is blocked by scoreboard), LSU_SRQ_SYNC_CYC (sync is in store request queue), and STCX_FAIL (a stcx instruction fails) events all have to do with synchronization. The LSU_LRQ_FULL_CYC event indicates that the load request queue is full and stalls dispatch.

Both the HV_CYC and EE_OFF events indicated increased kernel activity, and the difference in HV_CYC/CYC percentages accounts for just over 10% of all the cycles. The trace records for pseudojbb reflect both user and kernel mode execution; that is, the HPM counters continued counting when control enters the kernel. To determine if there was some underlying pathological Jikes RVM behavior that was causing increased kernel activity, we used the unix `truss` command to trace kernel calls. Other than `yield`, `_nsleep`, `thread_waitact`, and calls to the kernel HPM routines, there were no unusual kernel calls or increase in call activity that would explain the performance drop.

There is some indication that the IPC drop may be due to effective to real address translation (ERAT). The 128 entry IERAT and 128 entry DERAT are a cache for the 1024 entry TLB. We are investigating this hypothesis more thoroughly.

This anomaly illustrates the difficulty with determining performance anomalies with HPM information only. Deep microarchitectural, OS, and JVM knowledge is required to augment the HPM information.

5.5 Memory Wall Issues

We now use the `Performance Explorer` to explore how the scheduling of Java threads and garbage collection interact with the memory hierarchy. Our experiments use a semispace

[4]In a JIT configuration none of the AOS threads execute. In particular, there is no compilation thread; the cost of compiling a method is attributed to the application thread. Nevertheless, in a JIT configuration, the warehouse thread is interrupted only by the garbage collector and the infrequently executing finalizer thread. So once all the warehouse methods are compiled, every warehouse time slice runs for the full 10 millisecond quantum.

[5]Because the POWER4 has over two hundred events, but only 8 counters, this process had to be performed manually on many different trace files. In the future, this process could be incorporated into the `Performance Explorer` and automated.

garbage collector that divides the heap into two semispaces, *from* and *to*. The program allocates all objects in the *from* semispace. When garbage collector is triggered the collector copies all the live objects from the *from* space to the *to* space and switches the role of the *from* and *to* spaces. At the end of GC, all the live data is contiguous at the start of the newly-labeled `from` semispace.

Since Jikes RVM allocates code in the heap [3], the garbage collector moves around and compacts not just the data, but also the code that is not in the boot image (i.e., the compiler, standard libraries, etc.). The code is moved *through the data cache*. Thus after GC, references to instructions will miss in the instruction cache. The garbage collector does not move instructions in the boot image and thus after GC, references to these instructions may or may not miss (depending on how much of the cache is flushed by the garbage collector).

For the experiments reported in this section, we executed pseudojbb with the semispace garbage collector using the default adaptive heap size policy. The average amount of live data at the end of GC was around 25–30 MB. Before GC, the total heap space (excluding the *to* space) was about 100 MB.

5.5.1 Interaction of Threads with Memory System

We used the `Performance Explorer` to understand the impact of references and miss rates to the different memory hierarchy levels for both data and instructions of each Java thread. In particular, we compute the miss rate to L1 data cache by dividing the number of load misses from L1 (LD_MISS_L1) by the number of load references to L1 (LD_REF_L1). Unfortunately, the POWER4 HPM support does not provide miss and reference events to the other levels in the memory hierarchy, but does provides access counts: the number of times data is accessed from a particular cache. Therefore, we compute the number of references to a memory hierarchy level as the sum of the number of accesses to this level and to all lower levels. We compute the number of misses at a higher level in the memory hierarchy as the summation of the references at lower levels. The miss rate for a particular level is computed as the misses divided by the references to that level. For example, POWER4 HPM provides DATA_FROM_X events to specify the number of times data is accessed from the X level in the memory hierarchy. Thus, the L3 miss rate is (DATA_FROM_MM) / (DATA_FROM_L3 + DATA_FROM_MM).

Table 4 presents the references and misses metrics for the L1 instruction and data caches. The `Thread` column identifies the Java threads. The `Cycles` column identifies the percentage of total cycles spent executing this thread. The `Records` column reports the number of trace records captured for this thread. The `IPC` column reports instructions per cycle. The `Instructions` column reports the average number of instructions executed for each time quantum the Java thread is executed. The next two columns labeled `L1 Instructions` report the references and misses to the L1

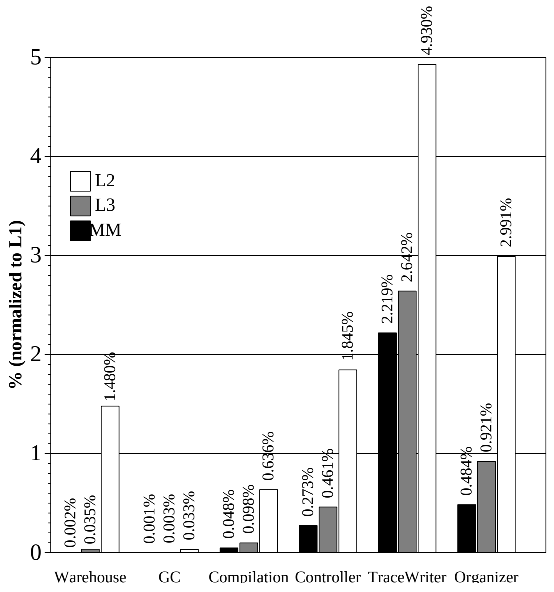


Figure 4: Instruction references to levels in the memory hierarchy normalized to all instruction references.

instruction cache. The last two columns report similar values for the L1 data cache. The reference and miss rate computed metrics for the other levels of the memory hierarchy can be computed from Table 4 and the subsequent Figures 4 – 7

Figures 4 and 5 illustrate the references to L2, L3, and main memory as a percentage of all references for both instructions and data. The horizontal axis is the Java threads and the vertical axis is the percentage of all references for a particular thread. For each thread, there are three bars, one for each memory hierarchy level: `L2`, `L3`, and `MM`. Figure 4 shows that the three Java threads (warehouse, GC, and compilation), which consume 79% of all cycles, have a 98% hit rate for L1 instruction cache, indicating excellent instruction locality. The daemon threads, which execute infrequently and for a short interval of time, have worse L1 instruction locality. This is expected, because of their infrequent execution, little or none of their instructions will be in the cache, and because of there short execution duration, the cache misses have a short interval over which to be amortized. Of the three daemon Java threads, TraceWriter stands out as having the worst instruction locality with almost 10% of its instructions not found in the L1 cache: half of the 10% is found in the L2 instruction cache. TraceWriter is the least frequently executed daemon thread.

For data references, illustrated in Figure 5, the story is better and remarkable consistent across all threads. At most 5% of data is not found in the L1 cache for any of the Java threads. Both the warehouse and GC threads have the best locality with less than 3% of the data not found in the L1 cache. This implies that the working sets of the Java threads fit into the 64KB data cache. As is expected, the lower the memory level, the fewer the references: L2 has fewer than 3.4% of all references, and main memory has less than 0.2% of all references

Thread	Cycles	Records	IPC	Instructions	L1 Instructions References	L1 Instructions Misses	L1 Data References	L1 Data Misses
Warehouse	59.12%	1995	0.481	11,261,031,705	6,080,171,921	89,966,131	5,885,067,463	461,297,214
GC	13.96%	11	0.524	2,885,482,384	1,447,776,154	482,694	1,113,092,487	281,135,697
Compilation	6.46%	271	0.565	1,531,568,783	905,536,702	5,757,690	769,831,082	89,280,559
Controller	0.11%	141	0.212	11,867,348	8,730,297	161,096	8,496,326	835,170
TraceWriter	0.05%	82	0.097	2,411,860	1,395,686	68,802	1,599,610	447,691
Organizer	0.03%	141	0.166	2,037,304	1,322,770	39,568	1,307,913	145,696

Table 4: L1 data and instruction cache references and misses for different Java threads.

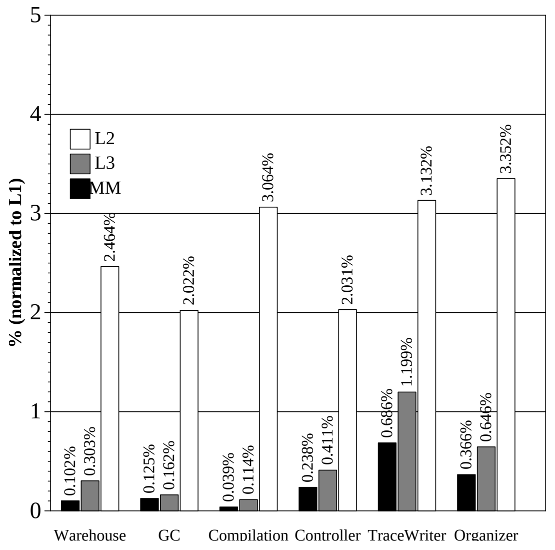


Figure 5: Data references to levels in the memory hierarchy normalized to all data references.

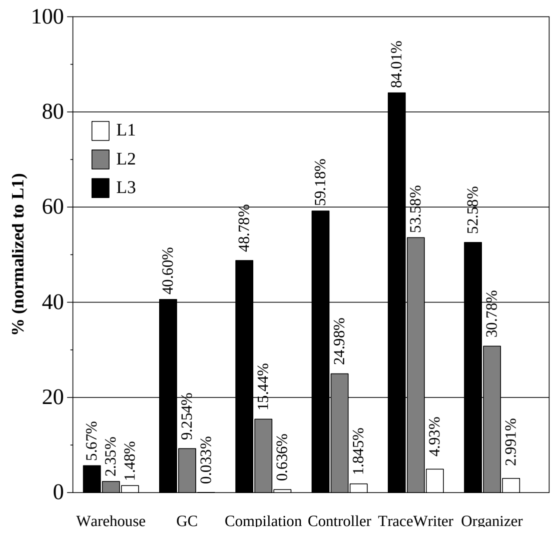


Figure 6: Instruction miss rates to levels in the memory hierarchy normalized to L1.

across all the Java threads.

Figures 6 and 7 illustrate the miss rates for the different levels of the memory hierarchy for both instructions and data. The figures illustrate that both the instruction and data L1 cache miss rates are small, less than 5% for instructions and less than 3.2% for data for all the Java threads. Furthermore, the figures illustrate that the L3 is of little help for all, but warehouse instructions: the L3 miss rate ranges from 33.57% to 84.01%. In general, the L2 miss rate is better for data than instructions. However, because the references to main memory and L3 are low, the high miss rates for L2 and L3 are not of a big concern.

Figure 6 shows that GC incurs a 0.033% instruction miss rate in L1. There are two reasons for this: (i) GC runs in uninterruptible mode where other Java threads cannot preempt its execution and evict its working set from the cache; and (ii) GC spends most of its time in a small inner loop which can easily fit in the cache. The instruction miss rates for the GC thread in L2 and L3 are not really relevant to performance since GC makes so few instruction references to L2 and L3.

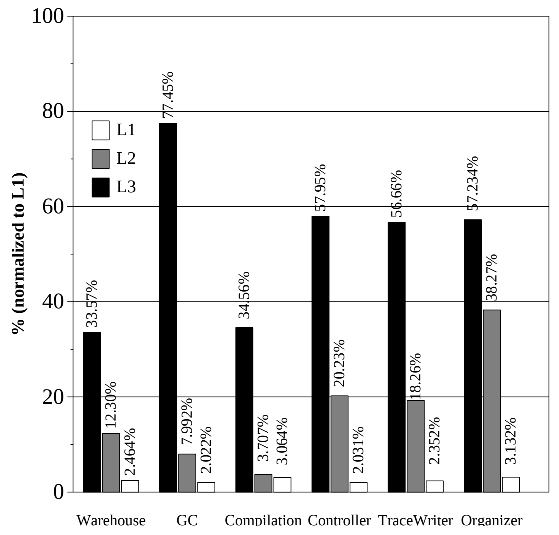


Figure 7: Data miss rates to levels in the memory hierarchy.

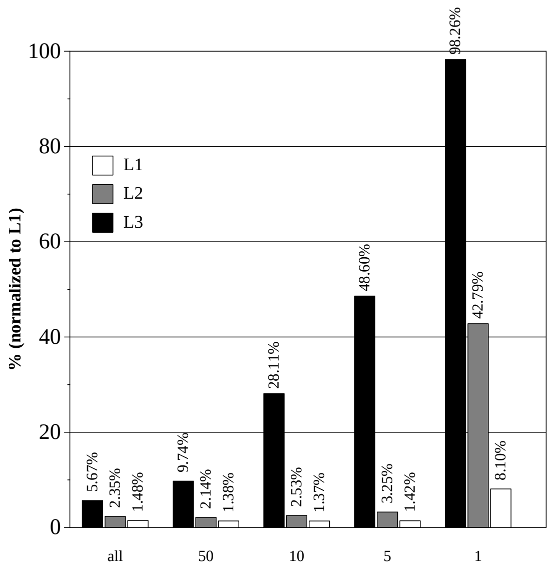


Figure 8: Instruction miss rates after a GC.

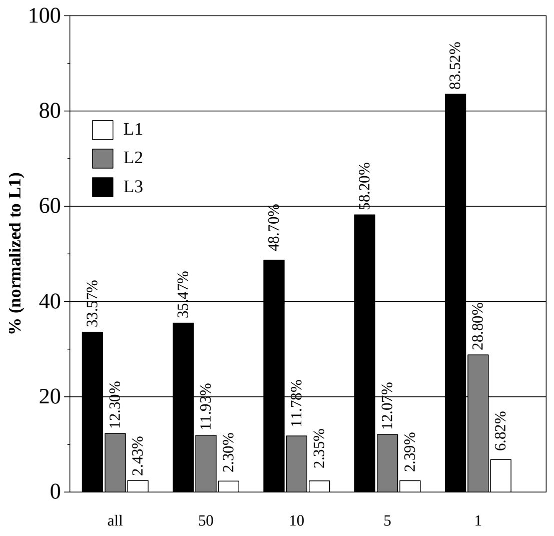


Figure 9: Data miss rates after a GC.

5.5.2 Effect of GC on Cache Performance

Because the garbage collector moves data and code, it can significantly affect the memory system behavior of the code that subsequently executes. To investigate this effect, we used `Performance Explorer` to extract miss rates for the first (1), the first five (5), the first ten (10), and the first fifty (50) warehouse trace records immediately after a GC. Figures 8 and 9 present this data for instructions and data, respectively. As a reference, the `All` bars present the overall cache miss rate of the warehouse thread.

From Figures 8 and 9 we see that both the instructions and data suffer increased cache misses immediately after a garbage collection. The misses in the L1 and L2 caches stabilize in less than 5 trace records. The much larger L3 cache takes even longer than 50 trace records to stabilize. The percentage of all references that are made to L2 is less than 0.30% for data and less than 0.05% for instructions for all but the first trace record after a GC. The percentage of all references that are to L3 is less than L2. The increased miss rates after a GC are also reflected in the IPC: the IPC in the first and first five trace records immediately following a GC are 0.069 and 0.357. In contrast the stable state IPC is 0.429.

These results clearly indicate that GC significantly degrades the memory performance of the application and it can take a significant amount of time, as indicated by the number of trace records, after a GC before the working set of the application is in the caches again. In future work we will investigate how other GC algorithms behave in this regard.

5.6 Discussion

We found that having a tool to manipulate the HPM data is indispensable. With over sixty thousand events in one 30-second execution of pseudojbb, some kind of tool is required to organize the HPM data. Surprisingly we found that, in many cases, visualization alone was not sufficient to detect trends. Alternatives, such as selecting subsets of trace records and computing average metric values over the subset, were required.

Because the POWER4 has only 8 counters, multiple runs of pseudojbb were required to collect traces for all the HPM events. The `Performance Explorer` is currently designed to work on one trace file at a time. Therefore using the `Performance Explorer` to understand trends across all the HPM events is rather tedious as a new window has to be opened for each trace. Extending the functionality of the `Performance Explorer` to perform more complex computations and support for better report generation would be helpful.

The performance degradation before GC demonstrates why traces for all HPM events are required as it is difficult to always know a priori what subset of HPM events are needed.

It took us a while to determine the correct computed metrics to explore the performance issues related to memory latency. One difficulty was understanding how HPM events can be combined to compute metrics. In general, the HPM data are part of the puzzle and getting the complete picture may require additional information or experiments. We found that the HPM data can help to determine what additional information is required. For example, we used JIT configurations to understand the IPC improvement over time for the adaptive configuration of Jikes RVM.

6 Related Work

This section surveys related work on software tools that collect, analyze, and visualize hardware performance counter data and other performance-related work for Java.

6.1 Accessing Hardware Counters

Several library packages provide access to hardware performance counter information, including the HPM toolkit [15], PAPI [11], PCL [9], and OProfile [26]. These libraries provide facilities to instrument programs, record hardware counter data, and analyze the results. We extend the functionality of existing libraries to obtain hardware performance data in a virtual machine. Specifically, we extend Jikes RVM to collect thread-specific, temporally fine-grained hardware counter data in an SMP environment.

The Digital Continuous Profiling Infrastructure provides a powerful set of tools to analyze and collect hardware performance counter data on Alpha processors [5]. The system includes tools to collect accurate profile data with very low overhead and to analyze the profile data using many performance metrics. The system works on unmodified executables on multiprocessors and collects time-based hardware counter samples of program counter values. VTune [32] and SpeedShop [35] are similar tools from Intel and SGI, respectively. Our work differs in that we are interested in correlating the hardware counter data to high-level program constructs, such as Java threads, to distinguish the effects from the VM and user applications, in an SMP environment in a temporal manner.

Ammons et al. [4] correlate hardware performance counter information to frequently executed program paths. They use flow- and context-sensitive data-flow analysis techniques to collect hardware counter data along program paths instead of just individual statements or procedures. Although this provides fine-grained information, the overhead of recording hardware counter data along the paths increases runtime by an average of 70%. The overhead of collecting and storing our HPM trace files is less than 2% in our VM.

IBM's Performance Inspector [30] 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. The package contains tools for sampling-based profiling (any performance counter can be used to trigger sampling), manual instrumentation-based profiling (measuring per-thread time/performance counts), and exception-based basic block profiling. Our work profiles several performance counters at once, does not require instrumentation by hand, and is tightly integrated with our VM, resulting in access to information about the runtime system (like compilation and garbage collection).

6.2 Profiling Java Workloads

The Java Virtual Machine Profiler Interface (JVMPI) defines a general purpose mechanism to obtain profile data from a Java VM [31]. JVMPI 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 measures the architectural level performance of Java applications.

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. [13] evaluate the performance of a Java implementation of the TPC-W benchmark and compare the results to SPECweb99 and SPECjbb2000. The TPC-W benchmark models an online bookstore, and the Java implementation is a combination of Java Servlets that communicate with a database system using JDBC. Cain et al. use hardware counters to measure the performance of the entire benchmark on an IBM multiprocessor with eight RS64-III processors. They also use simulation to experiment with new architectural features. Our infrastructure enables us to distinguish the performance between the various threads of the VM and application, on different processors, at regular time intervals.

Luo and John [21] evaluate SPECjbb2000 and VolanoMark on a Pentium III processor using the Intel hardware performance counters. Seshadri, John, and Mericas [28] use hardware performance counters to characterize the performance of SPECjbb2000 and VolanoMark running on two PowerPC architectures. The focus of both studies was to compare Java server applications to the SPECint2000 benchmarks, which are written in C. They obtain aggregate counter information over a significant portion of the benchmarks running a single processor, whereas we gather hardware performance data on multiple processors on a per thread basis at regular intervals.

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. Karlsson et al. use the hardware counters to measure coarse-grained events. The results are for the execution of the entire benchmark, and they do not distinguish between the VM and the application. Our infrastructure enables us to obtain fine-grained performance measurements information in real time.

Dufour et al. [16] 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 like array or pointer intensiveness, degree of polymorphism, allocation density, degree of concurrency, and synchronization. Our work analyzes workload characteristics on the architectural level, covers both application, library and the virtual machine behavior, and investigates behavioral patterns over time.

6.3 Statistical Performance Analysis

Recent work uses statistical techniques to analyze performance counter data. Eeckhout et al. [17] 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. We visualize behavior over time, require no instrumentation, and allow the analysis of any kind of derived metric.

6.4 Performance Visualization

Mellor-Crummey et al. [23] 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, whereas our focus is attributing them to processors and threads. They provide only metrics aggregated over the complete runtime. We show the value of metrics over time, which is important for understanding the application behavior in a virtual machine with a rich runtime system.

Miller et al. [24] 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. [34] introduce extensions to Paradyn to support the instrumentation of multithreaded applications. Our infrastructure contains full support for gathering hardware performance counters and is tightly integrated with the Java virtual machine's thread scheduler, which allows us to gather accurate performance measures for the complete system with very low overhead.

Zaki et al. [36] 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. [33] is also capable of correctly tracing multithreaded programs. We focus on tracing a single process on one SMP computer. Instead of tracing communication activity and user-defined program states of MPI (Message Passing Interface) programs, we gather and visualize the hardware performance of Java applications on a virtual machine.

Pablo, introduced by Reed et al. [27], is another performance analysis infrastructure focusing on parallel distributed 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. [14] 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. We focus on low overhead, fully automatic tracing of temporal data for a Java virtual machine and application. Our visualizations provide detailed information about the hardware performance and the behavior of the virtual machine.

7 Future Work

This paper uses the `Performance Explorer` to explore the performance behavior of one pseudojbb warehouse thread on one virtual processors. We are interested in performing additional experiments with more warehouses on one or more virtual processors, as well as exploring other benchmarks, to determine how the POWER4 HPM data may help to understand application behavior.

To provide temporal data, the system currently uses the context-switching among Java threads as the delimiter for counting intervals. Because the stop-the-world garbage collector prevents thread switching while it is executing, the system views a garbage collection, which can be longer than 10ms, as a single trace record. Because it is desirable to provide finer granularity for VM operations that disable context-switching, we plan to explore the addition of a separate trigger for capturing HPM information that will be in effect even when thread switching is disabled.

The adaptive optimization system of Jikes RVM's uses a cost/benefit model to determine which methods of an application should be optimized and at what optimization level [6]. We can leverage this system to attempt to selectively capture critical computations of the code in an automatic manner. Following in the direction of Arnold et al. [7], we could capture detailed profile information for only those methods that execute often. Specifically, we can instruct the optimizing compiler to automatically insert VM calls to partition the most time consuming computations and capture HPM information for these computations.

In this paper we explore using interactive visualization to drive performance analysis, which enables one to see large- and small-scale patterns, and to look at the data from any desirable perspective. A complimentary approach is to use statistical analysis to reduce the dimensional space. This is a worthwhile approach that we plan to explore.

8 Conclusions

We describe a system for exploring the low-level behavior of Java applications. Our system generates traces containing data obtained from hardware performance monitors and provides an interactive graphical user interface to explore the data. Although prior work presents tools for accessing hardware performance monitors, our work is unique in that it correctly attributes behavior to Java threads in a multithreaded system running on an multiprocessor. Our work is implemented in the context of Jikes RVM.

We demonstrate the usefulness of our tools by applying them to understanding the behavior of pseudojbb, a variant of the SPECjbb2000 benchmark. We demonstrate that our tools are able to identify as-yet unknown performance characteristics of the benchmark and are able to guide us in understanding the reasons for the observed performance characteristics. This understanding is essential for designing new high-payoff optimizations and for tuning applications and run-time systems for the best performance.

9 Acknowledgments

The authors thank Laureen Treacy for proofreading earlier drafts of this work. Amer Diwan and Matthias Hauswirth were partially supported by NSF grants CCR-0085792 and CCR-0133457. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

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

[3] 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.

[4] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *ACM SIGPLAN Notices*, 32(5):85–96, May 1997. Published as part of the proceedings of PLDI'97.

[5] Jennifer M. Anderson, Lance M. Berc, 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.

[6] 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).

[7] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, November 2002. Proceedings of the 2002 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'02).

[8] Steve Behling, Ron Bell, Peter Farrell, Holger Holthoff, Frank O'Connel, and Will Weir. *The POWER4 Processor Introduction and Tuning Guide*. Redbooks. IBM Corporation, International Technical Support Organization, 2001.

[9] Rudolf Berrendorf, Heinz Ziegler, and Bernd Mohr. PCL - the performance counter library. http://www.fz-juelich.de/zam/PCL.

[10] 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.

[11] 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.

[12] Michael G. Burke, Jong-Deok Choi, Stephen Fink, David Grove, Michael Hind, Vivek Sarkar, Mauricio J. Serrano, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño dynamic optimizing compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, June 1999.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *International Symposium on Code Generation and Optimization*, pages 253–264. IEEE Computer Society, 2003.

[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] 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.

[22] Cathy May, Ed Silha, Rick Simpson, and Hank Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 1994.

[23] 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.

[24] 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.

[25] Jeffrey C. Mogul and Anita Borg. The effect of context switches on cache performance. In *Proceedings of Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (Santa Clara, CA)*, pages 75–84, 1991.

[26] Oprofile. http://oprofile.sourceforge.net, 2003.

[27] 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.

[28] 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.

[29] The Standard Performance Evaluation Corporation. SPEC JBB 2000. http://www.spec.org/osg/jbb2000, 2000.

[30] 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.

[31] D. Viswanathan and S. Liang. Java Virtual Machine Profiler Interface. *IBM Systems Journal*, 39(1):82–95, February 2000.

[32] Intel VTune performance analyzers. http://www.intel.com/software/products/vtune.

[33] 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.

[34] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. In *Principles Practice of Parallel Programming*, pages 49–59, 1999.

[35] 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.

[36] 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.