RC22794 (W0303-150) March 27, 2003 **Revised 05/15/2003**
Computer Science

IBM Research Report

A VM Infrastructure for Understanding the Hardware Performance of Java Applications

Peter F. Sweeney, Brendon Cahoon, Perry Cheng,
David P. Grove, Michael J. Hind
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Delhi - Haifa - India - T. J. Watson - Tokyo - Zurich

A VM Infrastructure for Understanding the Hardware Performance of Java Applications

Peter F. Sweeney Brendon Cahoon Perry Cheng David Grove Michael Hind

IBM T.J. Watson Research Center
{pfs,brendon,perryche,groved,hindm}@us.ibm.com

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.

In this paper we describe the design and implementation of extensions to a virtual machine to access hardware performance monitors. The goal of the extended virtual machine is to enable a low overhead mechanism to collect thread-specific, fine-grained temporal hardware performance information on a multiprocessor. We extend Jikes RVM, an open source research virtual machine for executing Java programs, to periodically collect performance monitor information. We correlate the performance data to each of the application and VM threads and demonstrate the utility of the VM extensions by measuring the memory performance of SPECjbb2000 on a Power4 multiprocessor. We measure the load latency of each thread, separating the application threads from the VM threads, report the processor on which they execute, and show how the load latency varies over time for one application thread. We also discuss future extensions to this infrastructure.

1. INTRODUCTION

Modern microprocessors provide hardware performance monitors (HPMs) that count the number of times an event occurs, such as cache misses, pipeline stalls, and cycles consumed. The counts of a particular event can provide insight into how the underlying hardware resource associated with that event is utilized. For synthetic benchmarks or scientific codes where the computation is CPU intensive and highly uniform, aggregate counts for the entire run of the benchmark can be sufficient to determine bottlenecks.

However, aggregate counts are unlikely to be sufficient for understanding the performance of Java applications for several reasons. First, because a virtual machine's rich runtime support utilizes the same hardware resources as the application, it is important to distinguish resource usage of virtual machine (VM) threads from those of the application. Second, Java applications are often written as a collection of threads. Since each application thread may have different functionality it is important to distinguish the corresponding resource usage. Third, the characteristics of even a single Java thread may vary over its lifetime, so even thread-specific aggregate counts may not reveal underlying performance characteristics. Finally, in a symmetric multiprocessor (SMP) environment, a single Java thread may migrate among several physical processors. Understanding how such a thread's hardware performance characteristics vary among these processors may be useful.

This paper describes an approach to understand the performance of Java applications using hardware performance monitors in an SMP environment. Specifically, we discuss how to extend the Jikes RVM, an open source research virtual machine for executing Java programs, to access PowerPC hardware performance monitors. The extensions correlate resource usage with the associated Java threads and the physical processors they run on. In addition to the application's threads, this correlation includes VM threads, such as the garbage collector, the optimizing compiler, and the adaptive optimization system. We present the design and implementation of the VM infrastructure extensions, and illustrate how they can be useful in characterizing the hardware performance of Java applications. The ultimate goal of this work is to map these characteristics back to source code that caused them. This paper is a step in that direction.

The contributions of this paper are as follows:

- the design and implementation of extensions to an existing open source VM to periodically capture information from hardware performance monitors in an SMP environment and correlate it to the corresponding Java thread;
- illustrations of how this enhanced VM can be used to understand the hardware performance of SPECjbb2000 running on a Power4 SMP.

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

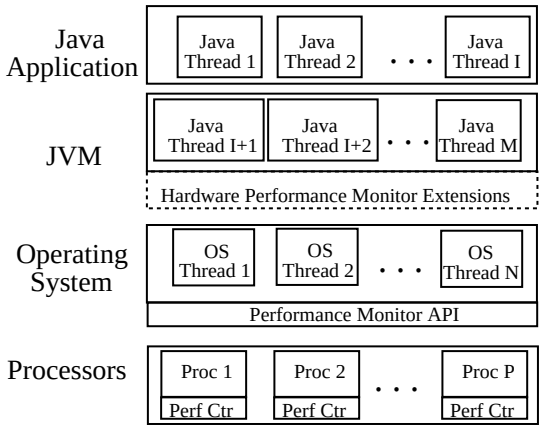

Figure 1: Architectural Overview

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 [12] 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, and Linux/PowerPC platforms. This section briefly provides background on a few relevant aspects of Jikes RVM. More details are available at the project web site [12] and in survey papers such as [1, 9, 5].

Because Jikes RVM itself is implemented in Java [2], 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, several virtual machine subsystems such as the garbage collector and adaptive optimization system [5] perform most of their work on dedicated Java threads. By gathering per-Java-thread HPM data we can easily separate the behavior of these VM daemon threads from the "normal" threads actually created by the application. However, our infrastructure currently does not discriminate between application, library, or VM code that executes on the same Java thread (for example, when an application method calls a system service such as allocation or dynamic type checking).

Motivated by a desire to efficiently support type accurate garbage collection when executing highly multi-threaded Java programs, an early design decision in Jikes RVM was to implement a layer of M-to-N threading, 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. We call the pThreads *virtual processors*, as each of them represents an execution resource (a virtual CPU) that the virtual machine can use to execute Java threads.

An integral part of Jikes RVM's threading system is compiler-supported quasi-preemptive scheduling. The compilers 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 simply needs to set the flag and wait 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.

2.2 AIX Hardware Performance Monitors

Most implementations of modern architectures (e.g. PowerPC Power4, IA-64 Titanium) 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 implementations of PowerPC (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 events and to begin counting the events. 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 [11].

Some processors provide more sophisticated facilities to access HPM data. These facilities include sampling and threshold mechanisms. The sampling mechanism counts hardware events and reports the values at regular time intervals. The threshold 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 to the software. The Power4 architecture provides sampling and threshold 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 traces containing thread-specific, temporally fine-grained HPM data in an SMP environment. The section begins by enumerating the design goals for the infrastructure. We then describe the trace record format and highlight 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 what really happened during 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 Each virtual processor is assigned a unique ID. Although this is redundant information (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 Each Java thread is assigned a unique ID by Jikes RVM. This field contains the ID of the thread that was executing during the measurement period. The ID is negated if the thread yields before its scheduling quantum expires.

Real Time 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) [15].

Real Time Duration 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 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 will be 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 CPUs. A secondary use of this data is to detect that the OS has scheduled other pThreads on the CPU 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 CPU 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 also contains marker records. We provide a mechanism, via calls to the VM, for a Java thread to specify points in an application's source file that when executed will place a marker record in the trace file. A marker record contains an arbitrary string. These marker trace records allow the programmer to post mortem 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, provides a mapping from counter to event name, and provides a mapping from thread ID to thread name. The number of counters and the mappings provide a means 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 both accumulated into per virtual processor and per Java thread summaries and written into a per virtual processor trace record buffer. Each virtual processor has two dedicated 4K trace record buffers. Trace records for a virtual processor are written into an active buffer. When the buffer is full, the virtual processor writes to the other buffer and signals a dedicated Java thread, called a *trace writer* thread, to drain the full buffer into a file. There is one trace writer thread per virtual processor.

By alternating between two buffers, we continuously gather trace records with low overhead. By having a dedicated Java thread drain the buffer, we avoid directly perturbing the behavior of the other threads in the system. It 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 are currently available in Jikes RVM [12] from the head of the source tree.

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, if a trace record is written every 10 milliseconds and a record size is 88 bytes, then a virtual processor writes 8.8 Kbytes to disk every second.

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 other 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. SAMPLE USAGE

This section demonstrates how our extensions to Jikes RVM are used to understand the load latency behavior of the SPECjbb2000 benchmark when run on a 24-way p690 Power4 SMP with a clock rate of 1.1 GHz. SPECjbb2000 emulates the middle tier of a 3-tier warehouse order system where each warehouse is modeled as a Java thread.

The Power4 is a 64-bit microprocessor that contains two processors on each chip. The Power4 contains three cache levels. The L1 data cache is 64KB, two-way set associative, contains 128B cache lines, and has a latency of about 3 cycles. The unified on-chip L2 cache is 1.5MB, eight-way set associative, contains 128B cache lines, and has a latency of about 12 cycles. Two processors share an L2 cache. The unified off-chip L3 cache is 32MB, eight-way set associative, contains 512B cache lines, and has a latency of about 120 cycles. The latency to main memory is about 350 cycles.

In our experiment, we ran four warehouses on four processors, resulting in four trace files, one for each virtual processor. The hardware performance monitor counters were collected when both user and kernel code was executed over a (measurement) period of 40 seconds for all four processors, using the mechanism described in Section 3.3 to identify the trace records for the 40 second period. A total of 27,951 trace records were collected on the 4 processors, consuming 2.5 Mbytes.

Twenty-one Java threads were created during the execution, five by the application and sixteen by the VM. The application threads are the *main* thread (Main) and four *warehouse* threads (WH) that perform most of the application's computation. The VM creates a *garbage collection* thread (GC) for each virtual processor, so that stop-the-world collection can occur in parallel. The VM also creates an *idle* thread for each processor, which helps load balance Java threads when a virtual processor is idle. As mentioned earlier the VM also creates a trace writer thread (TW) for each virtual processor to transfer trace records from the buffer to a file. The last four threads are related to the adaptive optimization system [5]. The *optimization* thread (OPT) performs optimized compilations of methods selected by the *controller* thread (Cont), which processes online profile data recorded by the *method sampler* thread (MS). The *adaptive inlining* thread (AI) processes dynamic call site samples and uses this information to suggest further inlining candidates.

Figure 2 gives the breakdown of cycles executed by each of the twenty-one threads. The vertical axis measures the percentage of total cycles across all virtual processors for the measurement period, and the horizontal axis identifies the Java threads. A bar represents a thread that has executed on either one or more virtual processors. A multi-shaded bar indicates that the thread has executed on multiple virtual processors, where each virtual processor is identified by a different shade. Most of the cycles (72%) are spent in the applications warehouse threads (WH1, WH2, WH3, WH4), where each warehouse runs on a distinct virtual processor. About 23% of the cycles are spent in the garbage collection threads (GC1, GC2, GC3 and GC4). The only other thread that takes a noticeable number of cycles (3%) is the optimization thread (OPT). The optimization thread is not bound to a particular processor; it executes on all processors as indicated by the multi-shaded bar. The other threads' combined cycles represent a small fraction ($< 3\%$) of execution time.

Load latency is an important metric of performance be-

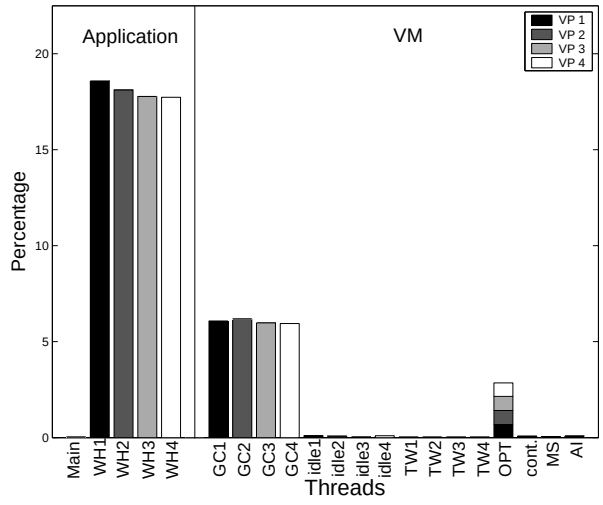

Figure 2: The number of cycles executed as a percentage of total cycles.

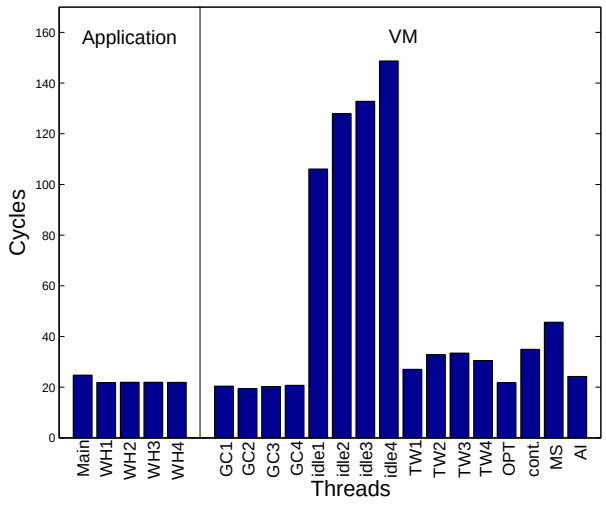

Figure 3: Average load latency.

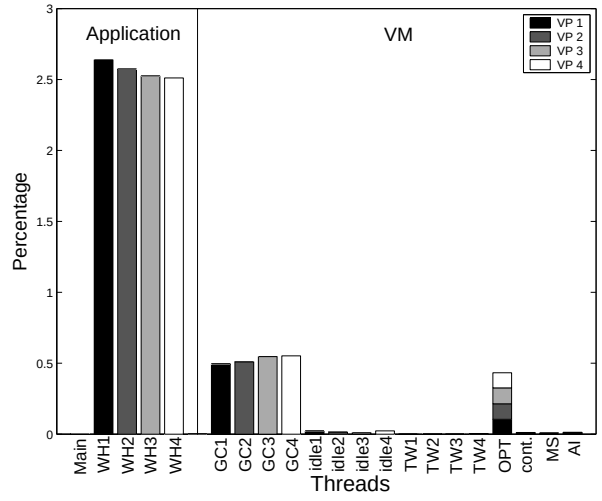

Figure 4: Estimated Load latency cycles as a percentage of total cycles across all processors.

cause it indicates how well an application exploits data locality. Using the extensions described in this paper, we measure how load latency is distributed across the Java threads, and how load latency varies over time for any of these threads. The Power4 performance monitors do not compute the load latency directly, but we can use the LRQ_S0_ALLOC and LRQ_S0_VALID performance counters to estimate the load latency. The LRQ_S0_ALLOC event counts the number of times the first entry in the load reorder queue is allocated. The LRQ_S0_VALID event counts the number of cycles that a load remains in the first entry of the load reorder queue. The Power4 has 32 entries in the load reorder queue. The estimated load latency is LRQ_S0_VALID divided by LRQ_S0_ALLOC. The estimated load latency during our measurement period was 21.72 cycles; that is, on average a load took almost 22 cycles. In the rest of the paper we use the term load latency to mean estimated load latency.

Figure 3 illustrates the average load latency for the different threads. The vertical axis is the average load latency in cycles and the horizontal axis identifies Java threads. The height of a bar is the average load latency across virtual processors for each thread. The garbage collector has the lowest average load latency (about 20 cycles). We believe that this better than average locality is due to the fact that the garbage collection thread is not interrupted by other Java threads when it runs; although, it may be interrupted by OS kernel threads. This low latency is also desirable because approximately 23% of the cycles are spent in the GC threads.

The application warehouse threads and optimization thread have the next lowest average cycles per load (just under 22 cycles). Because most of the cycles are spent in the warehouse threads, this is also desirable. The idle threads have poor locality. Each processor has an idle thread that helps load balance Java threads. When a processor has no Java thread to execute, the idle thread is scheduled and runs for a short duration, making several system calls. Fortunately, little time is spent in the idle threads, so the impact of the poor latency is minimal. Finally, the other threads all have worse locality than the application and GC threads. The high average load latency may be explained because they are infrequently executed for short durations of time. This leads to a high probability that the data they access will not be in cache. For example, the method sampler (MS) thread processes a buffer of sampled methods identifiers approximately every 100 milliseconds. The trace writer (TW) and adaptive inlining (AI) threads have similar behavior.

Figure 4 illustrates which threads consume the most load latency cycles. The vertical axis measures the percentage of total cycles across all virtual processors for the measurement period, and the horizontal axis identifies the Java threads. Each virtual processor is identified by a different shade. Load latency cycles for the first entry in the load request queue are about 13% of the total cycles. [2] This percentage is computed as the summation of the height of all the bars. As expected, the threads that execute the longest contribute the most load latency cycles.

To understand how load latency varies as the application executes, Figure 5 illustrates the average load latency for a warehouse over time. The horizontal axis is real time

[2]The cycles consumed by the other load request queue entries are not available from the POWER4 hardware performance monitors.

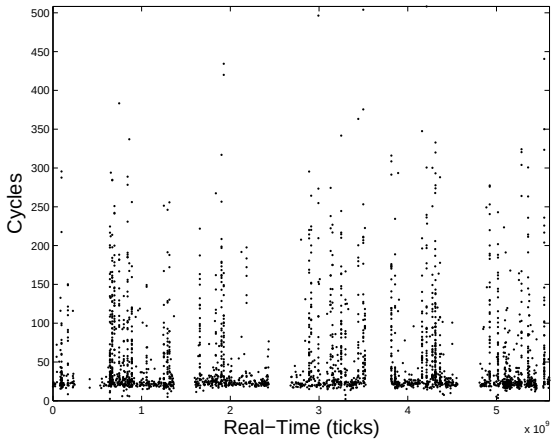

Figure 5: The load latency for a warehouse over time. Gaps indicate the execution of the GC thread.

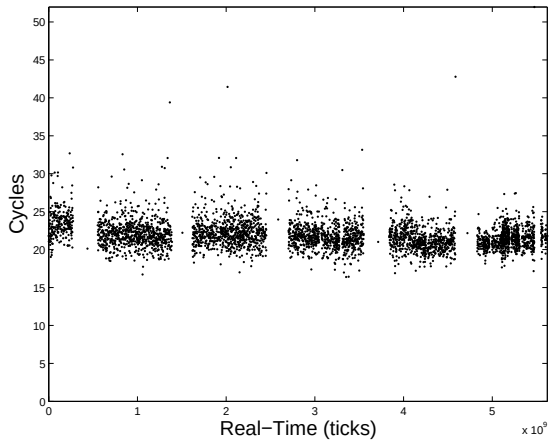

Figure 6: The load latency for a warehouse over time for those trace records that executed for at least 1 million cycles. Gaps indicate the execution of the GC thread.

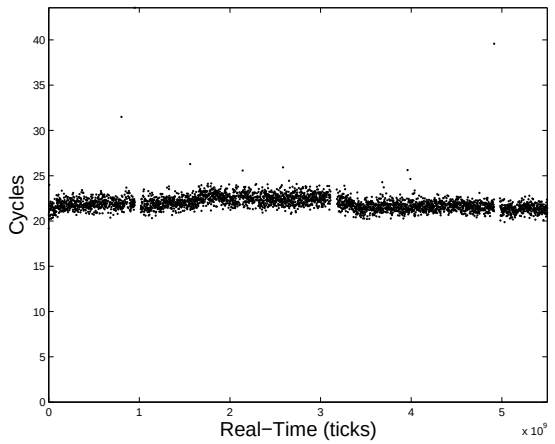

Figure 7: The load latency for a warehouse over time when SPECjbb2000 is run with only one warehouse. Gaps indicate the execution of the GC thread.

and the vertical access is load latency in cycles. Interestingly, the average load latency is not uniformly distributed, but has significant variation. Of the 6,640 trace records that are generated for this warehouse, 60.6% yield before the scheduling quantum of 10 milliseconds expires. Code inspection revealed that warehouse yields are due to synchronization. Synchronization has poor locality because a warehouse synchronizes on a global object, and as is the case when a global variable is accessed by multiple processors on a multiprocessor, the data has to be updated across caches. The five gaps in the graph represent the duration of different stop-the-world garbage collections. During a garbage collection, all non-GC threads are stopped waiting for the collection to complete. A garbage collection thread is uninterruptible; that is, it is not swapped out by Jikes RVM until it completes execution.

Figure 6 illustrates the average load latency for the same warehouse over time when considering only those trace records that execute for at least 1 millisecond or 10% of the scheduling quantum. Again, the gaps in the graph represent that the garbage collector is executing. In this graph, the average load latency is more uniformly distributed within a range of 20 to 25 cycles. With the filtering, only 45% of the original trace records are included and of these, 44% yield due to synchronization. We hypothesize that when a thread runs for longer than 1 millisecond, either the thread's progress does not require synchronization, or the cost of the synchronization is amortized across the additional loads of local data.

Figure 7 illustrates the average load latency for a warehouse over time when only one warehouse is run on one processor. No filtering is done for this graph and yet there are few outlying points. Since the single warehouse always successfully obtains a lock without contention, this supports our hypothesis that the outlying points are due to synchronization-induced yields. In addition, we note that the estimated load latency is distributed over the narrower range of about 20 to 23.

5. RELATED WORK

We briefly describe existing software tools that collect and analyze hardware performance counter data. We also present related work that uses hardware counter data to understand Java middleware and server applications.

5.1 Accessing Hardware Counters

Several library packages provide access to hardware performance counter data, including the HPM toolkit [11], PAPI [8], and PCL [7]. 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 [4]. 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 collects time based hardware counter samples of program counter values. The infrastructure works on unmodified executables running on multipro-

cessors. VTune [18] and SpeedShop [19] are similar tools from Intel and SGI machines, 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. correlate hardware performance counter information to frequently executed program paths [3]. 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.

5.2 Profiling Java Workloads

The Java Virtual Machine Profiler Interface (JVMPI) defines a general purpose mechanism to obtain profile data from a Java VM [17]. The goal of JVMPI is to enable tool vendors to create profiling tools without changing the VM. JVMPI supports CPU time profiling for threads and methods, heap profiling, and monitor contention profiling. JVMPI performs CPU time profiling using either statistical sampling or code instrumentation. 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. evaluate the performance of a Java implementation of the TPC-W benchmark and compare the results to SPECweb99 and SPECjbb2000 [10]. 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 evaluate SPECjbb2000 and VolanoMark on a Pentium III processor using the Intel hardware performance counters [14]. Seshadri, John, and Mericas use hardware performance counters to characterize the performance of SPECjbb2000 and VolanoMark running on two PowerPC architectures [16]. 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. Our infrastructure obtains hardware performance data on multiple processors on a per thread basis at regular intervals.

Karlsson et al. characterize the memory performance of Java server applications using real hardware and a simulator [13]. 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.

6. FUTURE WORK

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 current infrastructure 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.

Other short-term future directions include 1) determining the distribution of loads to the different levels in the memory hierarchy, 2) tracking other hardware events, such as i-cache misses, synchronization, and branch misprediction, on SPECjbb2000 and other benchmarks, 3) developing a graphical tool to navigate the trace files, and 4) exploring how other HPM mechanisms, such as the threshold mechanism described in Section 2.2, can be utilized.

The ultimate goal of this work is to correlate HPM events back to the source code that caused them. The work described in this paper takes a step in that direction by capturing the event information over time and attributing it to Java threads and processors. However, this information may still not be sufficient for a programmer or automatic optimizer to address the underlying performance problem. In addition to the approach of using calls to the VM to partition logical computations in the source as described in Section 3.3, we plan to investigate the following approaches to further this correlation:

- Build on the current automatic aspect of the system to associate additional runtime information beyond the current thread and virtual processor with a trace record. One option is to exploit the availability of the thread's runtime stack. Jikes RVM provides convenient mechanisms for capturing the full call stack, or a subset of it, including the source code for each call site.

- Jikes RVM's adaptive optimization system uses a cost/-benefit model to determine which methods of an application should be optimized and at what optimization level [5]. 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 [6], 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.

7. CONCLUSIONS

This paper describes how an existing open source VM is extended to help understand the hardware performance of Java applications. The extensions provide the ability to attribute hardware events to any Java thread, application or VM, that executes on one or more processors. In addition to aggregate information, the infrastructure provides a temporal account of a hardware resource, allow-

ing one to understand how a Java thread that executes on one or more processors uses a hardware resource over time. We present the usage of the infrastructure to understand how well the SPECjbb2000 benchmark exploits data locality. In particular, by exploiting the fine-grained nature of our traces and the use of a simple filter, we hypothesized and confirmed that thread-switching due to lock contention can cause spikes in the load latency.

8. REFERENCES

[1] 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.

[2] Bowen Alpern, C. R. Attanasio, Anthony Cocchi, Derek Lieber, Stephen Smith, Ton Ngo, John J. Barton, Susan Flynn Hummel, Janice C. Sheperd, and Mark Mergen. Implementing Jalapeño in Java. *ACM SIGPLAN Notices*, 34(10):314–324, October 1999. Published as part of the Proceedings of OOPSLA'99.

[3] 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.

[4] 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.

[5] 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. Published as part of the Proceedings of OOPSLA'00.

[6] Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online feedback-directed optimization of Java. *ACM SIGPLAN Notices*, 37(11):111–129, November 2002. Published as part of the Proceedings of OOPSLA'02.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

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

[16] 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.

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

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

[19] 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.