Dynamic Instrumentation of Threaded Applications[1]

Zhichen Xu   Barton P. Miller   †Oscar Naim

{zhichen,bart}@cs.wisc.edu, †onaim@us.oracle.com

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

†Oracle Corporation
1000 SW Broadway, Suite 1200
Portland, OR 97205

Abstract

The use of threads is becoming commonplace in both sequential and parallel programs. This paper describes our design and initial experience with non-trace based performance instrumentation techniques for threaded programs. Our goal is to provide detailed performance data while maintaining control of instrumentation costs. We have extended Paradyn's dynamic instrumentation (which can instrument programs without recompiling or relinking) to handle threaded programs.

Controlling instrumentation costs means efficient instrumentation code and avoiding locks in the instrumentation. Our design is based on low contention data structures. To associate performance data with individual threads, we have all threads share the same instrumentation code and assign each thread with its own private copy of performance counters or timers. The asynchrony in a threaded program poses a major challenge to dynamic instrumentation. To implement time-based metrics on a per-thread basis, we need to instrument thread context switches, which can cause instrumentation code to interleave. Interleaved instrumentation can not only corrupt performance data, but can also cause a scenario we call self-deadlock where an instrumentation code deadlocks a thread. We introduce *thread-conscious locks* to avoid self-deadlock, and *per-thread virtual CPU timers* to reduce the chance of interleaved instrumentation accessing the same performance counter or timer, and to reduce the number of expensive timer calls at thread context switches.

Our initial implementation is on SPARC Solaris2.x including multiprocessor Sun UltraSPARC Enterprise machines. We tested our tool on large multithreaded applications, including the Java Virtual Machine (JVM). We show how our new techniques helped us to speed up a Java native method by 22% and overall execution time of the JVM interpreting the AppletViewer driven by a game applet by 7%.

1 INTRODUCTION

Multithreading is a powerful technique to exploit parallelism on multiprocessors and to improve performance on uniprocessors by overlapping computation with I/O [3]. It provides a general-purpose solution for managing concurrency, and has been adopted by many of today's core applications, including database and web servers, Internet search engines, Java interpreters, web applications, irregular numerical applications, and graphical user interfaces. A multithreaded program, unfortunately, faces more obstacles to good performance than a sequential program. These obstacles include: competition for resources, synchronization, context-switching and non-overlapped I/Os. In response to these problems, we have extended the Paradyn performance measurement tools to monitor and analyze the performance of threaded applications. This paper describes the techniques that we used and initial results we obtained from applying these techniques to real programs.

We have the following goals in this research:

1. This work is supported in part by Department of Energy Grant DE-FG02-93ER25176, NSF grants CDA-9623632 and EIA-9870684, and DARPA contract N66001-97-C-8532. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

- Provide detailed information to locate performance bottlenecks in a threaded program, and to associate them with individual threads and groups of threads.
- Handle the dynamic nature of the environment, including frequent thread creation and deletion, and frequent insertion and removal of instrumentation (Paradyn allows users to insert and remove instrumentation during program execution).
- The instrumentation overhead for threaded programs should be comparable to that for non-threaded programs.
- Handle different thread packages, including user-level and kernel scheduled threads, and preemptive and non-preemptive scheduling.

As a result of this research, we can provide detailed performance information for a threaded program without using tracing or special compiling and linking of the application.

A major challenge in building a performance tool for threaded programs is to associate performance data with individual threads and doing so with acceptable overhead. We adopt a design, called *Same Instrumentation Code Multiple Data* (SICMD), in which all threads share the same instrumentation code, but each has its own private copy of performance counters or timers. This design greatly simplifies the instrumentation code and reduces locking to only a few global bookkeeping data structures. Avoiding locks in the instrumentation code makes it less intrusive.

A second challenge is to handle the asynchrony that can occur in multithreaded programs, such as thread preemption. Instrumenting context switches is necessary for several reasons, such as implementing time-based metrics. However, this can cause instrumentation code to interleave with the execution of a thread. When instrumentation code calls functions that use locks (such as used in a thread-safe C library), it might deadlock the thread. We call this scenario *self-deadlock*. We have devised a lock structure, called *thread-conscious locks,* to avoid self-deadlock. A similar problem can occur with Paradyn's mechanism for asynchronously triggering instrumentation code in the application (called an *inferior RPC).* We devised a safe inferior RPC to ensure that it will not interleave with other instrumentation code.

A third challenge is to minimize instrumentation cost when measuring time-based metrics on a per-thread basis. To measure the time spent by individual threads (i.e., to virtualize timers per thread), we need to turn timers on and off at thread-context switches. When context switching is frequent, this additional instrumentation can substantially slow down an application. We introduce per thread virtual timers, and implement other timers using the virtual timers. This approach also reduces the chance that interleaved instrumentation access the same timer structure.

In addition to the above technical contributions, there are important features of Paradyn that our tool gets for free. For example, Paradyn's *dynamic instrumentation* technology enables our tool to instrument an unmodified binary and running program. This allows us to monitor a multithreaded server that is already running.

Our initial implementation runs on SPARC Solaris2.x, including multiprocessor Sun UltraSPARC Enterprise machines. We have tested our tool on large multithreaded applications including the Java Virtual Machine (JVM), and we show how our new techniques helped us to speed up a Java native method by 22% and reduce overall execution time of the JVM interpreting the AppletViewer driven by a game applet by 7%.

In this paper, we will refer to *threaded Paradyn* for the new version of Paradyn that can instrument threaded programs, and *non-threaded Paradyn* for the previous version of Paradyn. We use the term *process* to refer to a kernel scheduled entity that has its own address space, light-weight processes (*LWP)* to refer to kernel-scheduled threads that execute in a process, and *threads* to refer to user-level scheduled threads of execution. There are many other uses of similar terms (e.g., Microsoft uses the terms process, thread, and fiber).

2 PARADYN BASICS

We briefly describe the basic characteristics of the Paradyn performance tool (more complete descriptions appear elsewhere [1, 2]). Paradyn is a parallel performance measurement tool that currently runs on SPARC (Solaris), Alpha (DEC UNIX), Power2 (AIX), and x86 (Solaris, NT, Linux) platforms. Paradyn uses dynamic instrumentation, a technology that allows instrumentation code to be inserted, changed, and removed from a running application. Beside standard performance metrics such as CPU and waiting times, Paradyn can instrument any system, hardware, or network activity that is visible in an application program's address space.

Paradyn provides two basic abstractions for performance data: *metric* and *focus*. A metric is a time-varying function that measures some aspect of an application's performance, such as CPU utilization or procedure call frequency. A focus is a component of a running application. Paradyn views a program as a collection of resource hierarchies that represent various elements of a program, such as code (modules, procedures), machines (processes, and now threads), or synchronization (messages, semaphores, locks). A selection of nodes in the resource hierarchy forms a focus. Tool users request Paradyn to collect metrics for the foci in which they are interested, and Paradyn dynamically instruments the program according to these criteria.

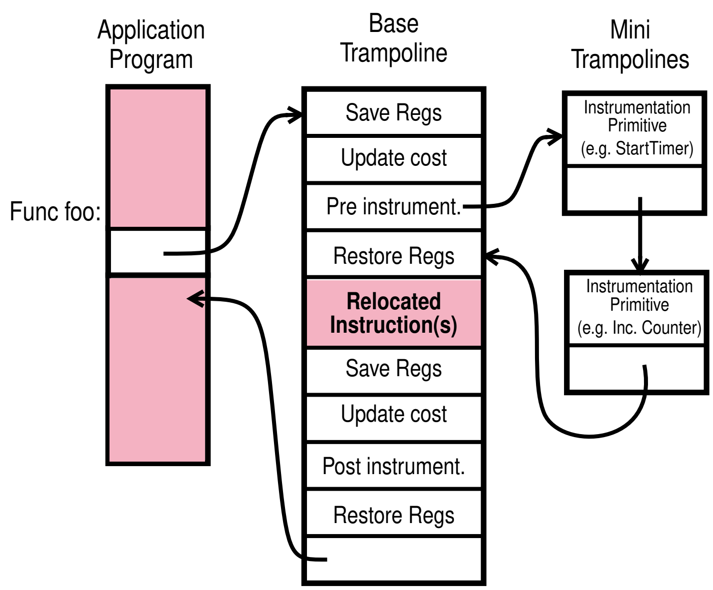

Figure 1: Trampoline Structure (Non-threaded Version of Paradyn)
The entry point of function "foo" has been instrumented.

In non-threaded Paradyn, instrumentation code and data (counters or timers) are allocated on a per process basis. The instrumentation code uses a trampoline structure (see Figure 1). For each runtime instrumentation request, Paradyn patches a jump to a *base-trampoline*, and relocates the instructions that were overwritten. Each base-trampoline has pre- and post-instrumentation sections. The pre-instrumentation section links all instrumentation code to be executed before the program point, and the post-instrumentation section links all instrumentation to be executed after the program point. Each linked item is called a *mini-trampoline*.

3 Instrumenting Threaded Programs

Our goal is to provide fine-grained information to locate performance bottlenecks in a threaded program and quantify them with respect to individual threads and groups of threads. We achieve this goal by adopting a design called Same Instrumentation Code Multiple Data (SICMD) that uses a single copy of instrumentation code and per-thread copies of performance counters and timers. We extended the resource hierarchy of Paradyn to include new resources for threads and thread-specific synchronization objects.

3.1 Same Instrumentation Code Multiple Data

In a multithreaded application, all threads share the same code, and modifying the application code affects all threads. We used same instrumentation code multiple data, where all threads share the same instrumentation, but each has its private copy of performance data. Performance metrics for group of threads are computed by aggregating measurements for individual threads.

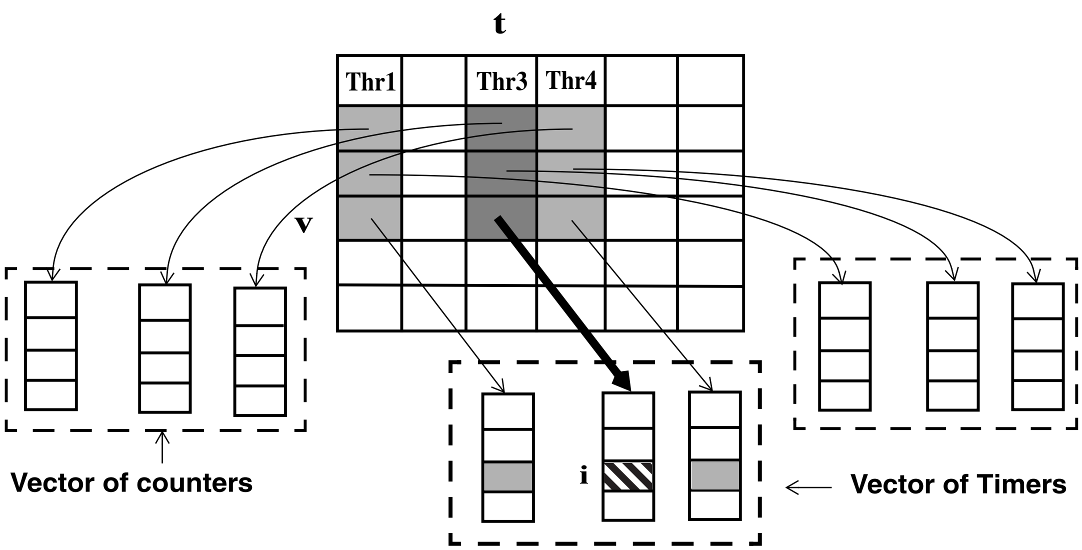

Figure 2: Thread Table

The main data structure (in the application's address space) is a matrix called the *Thread Table*. Each active thread has a column in the table. Table cells in each row point to a vector of counters or timers. Whenever a thread is created, a free column in the Thread Table is assigned to the newly created thread. For each metric-focus pair enabled, a row of counters or timers is allocated corresponding to a vector of counters or timers in the non-threaded Paradyn. A counter or timer is identified by [t, (v, i)]: thread t, entry i in vector v. For example, to measure CPU time spent by the thread `Thr3` in Figure 2, a timer will be allocated. It is identified by column t, row v, vector element i.

The trampoline structure for instrumenting threaded programs is shown in Figure 3. There is a new section, called the *MT Preamble,* that maps the thread ID to the address of the column in the Thread Table. The column address is stored in a register to be used by all mini-trampolines. In a mini-trampoline, an extra piece of code computes the counter or timer address based on the value of the register that holds the column address and (v, i) for the corresponding metric-focus pair.

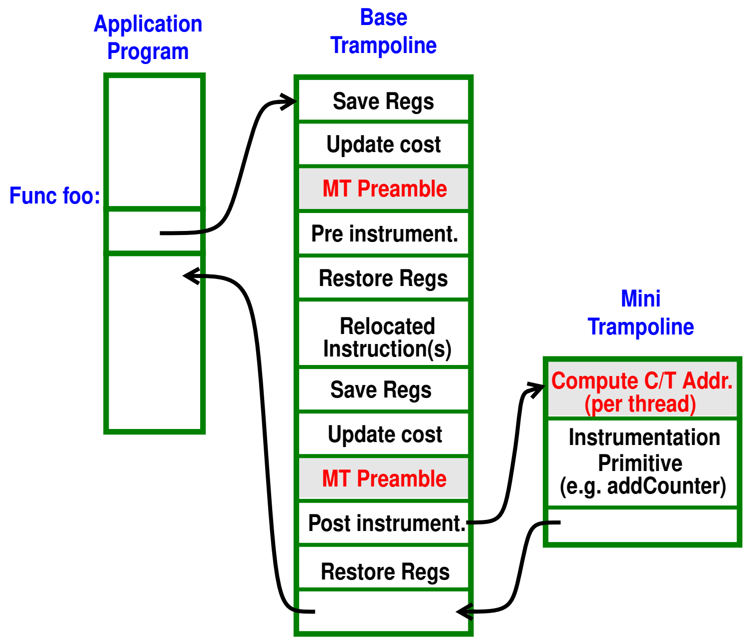

Figure 3: Trampoline Structure (with Threading)

New features are highlighted.

The advantages of this design are: (1) the trampoline structure is similar to the non-threaded version, (2) no locks are needed for the counter or timer structures, and (3) the address calculation for counters or timers is simple and efficient. The disadvantage is that some counter and timer slots may never been used.

To maintain the thread table, we instrument the thread creation and deletion routines. In addition to the Thread Table, we use three global data structures: a free list of columns in the Thread Table, a dictionary that maps a thread ID to the column index assigned to the thread, and a table of thread IDs of threads that have already exited. The dictionary is updated at each thread creation and deletion, and is used in each base trampoline to compute the column address. The table of deleted threads is used to tell if a newly created thread has already terminated (this can occur if a thread is extremely short-lived).

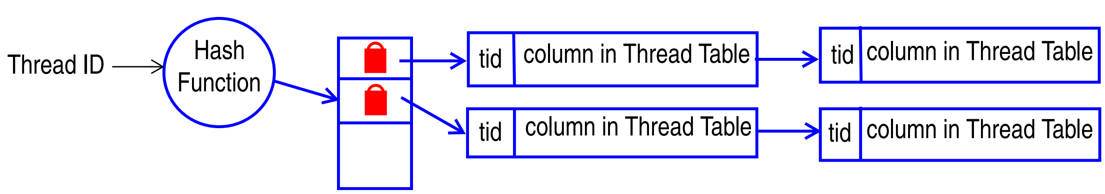

Figure 4: Low-Contention Data Structures

Since multiple threads can simultaneously access the global bookkeeping data structures, locks are needed to protect them. To reduce contention for locks, we implement the dictionary and the table for terminated threads as hash-tables shown in Figure 4. Each hash table bucket has its own lock, and only threads whose thread IDs hash to the same bucket could contend with each other. In our current implementation for SPARC Solaris2.x, we cache the column index assigned to a thread in a global register that is preserved across procedure calls (reducing table access).

Several technical difficulties arise from monitoring multithreaded programs that have already started to run. The most important one is to obtain information about threads that have already been created. Important information includes the stack base of a thread and the ID of the LWP on which the thread is currently mapped. The stack base is needed to walk the stack to ensure that no thread is executing the instrumentation code when we try to remove it. The LWP ID is needed to ensure that the timer metric is correct (see Section 3.3). We can add code to our instrumentation to detect the first time a thread executes an instrumentation code, or check the data structures maintained by the thread package for these information.

3.2 Dealing with Asynchrony

Because multiple threads can access global instrumentation data simultaneously, we use locks to protect the global bookkeeping data structures. Moreover, thread preemption can cause interleaving of instrumentation code. Consider the case when a thread is preempted while executing instrumentation code. The instrumentation installed in the thread context switch routines will interleave with the instrumentation that was preempted. If they both write to the same performance counter or timer, the performance data could be corrupted. Making the situation worse is that any use of locks in the instrumentation code has the potential to deadlock the application. There is an interesting scenario, which we call *self-deadlock.* This scenario occurs when instrumenting thread context switches, when instrumentation code in the context switch routine requests a lock, and could then deadlock if the thread has been granted the same lock.

Previous		**New**		**Return Value**
State	**TID**	**State**	**TID**	
HELD	t_1	HELD	t_1	SELF
HELD	t_2	HELD	t_2	NO
FREE	-	HELD	t_1	YES

Table 1: Thread-Conscious Lock

Thread t_1 requests tc-lock(l), where l may already be held.

To solve the self-deadlock problem, we introduced a lock structure called the *thread-conscious lock (tc-lock).* This lock has a field holding the ID of the thread that currently holds the lock and returns a special value if the lock acquisition routine detects that the thread already has been granted the lock. The instrumentation code tests the return value of the tc-lock to avoid self-deadlock, and allowing code to handle this case. Table 1 illustrates the logic of tc-lock.

In our current implementation, if a tc-lock returns `SELF`, we simply skip the instrumentation. As described in Section 3.1, we cache the column index assigned to a thread in a global register. This greatly reduces dictionary look-ups and the contention on locks. We have also introduced per-thread virtual timers to reduce the chance of the same timer data being accessed by interleaved instrumentation (see Section 3.3). A more thorough solution is to add extra information to the lock, counter and timer structures to effect communication between interleaved instrumentations to carry out appropriate actions.

3.3 Timer Issues

Another challenge is to minimize instrumentation cost when measuring time-based metrics on a per-thread basis. Threads are executed by LWPs. To measure the CPU (virtual) time of an individual thread, we must use the per-LWP timer kernel calls and instrument thread context switches to account for thread context switching and migration. To reduce the number of calls to expensive timer routines, we introduced per-thread virtual timers, one for each thread, and implement our performance timers using the virtual timers.

The initial LWP ID of a newly created thread is recorded in a virtual timer and updated at every thread context switch to account for any possible thread migration. We turn a virtual timer off when a thread is de-scheduled (this stops all performance timers for this thread) and turn it back on when a thread resumes execution. Implementing per-thread virtual timers reduces the number of calls to expensive timer routines, and reduces the chance that a timer structure gets accessed by interleaved instrumentation.

3.4 Inferior RPC

Paradyn uses a mechanism called inferior RPC to asynchronously trigger instrumentation code. Inferior RPC is used to start instrumentation, when execution has already past the point in which it was inserted. For example, to measure CPU for function *foo*, we insert start-timer code at the function entry and stop-timer code at the exit. If we are already executing in *foo* when the instrumentation request is made, we use inferior RPC to trigger the entry instrumentation to start the timer.

Paradyn implements inferior RPC by pausing the application, installing the inferior RPC code into the application's address space, and changing the program PC to the inferior RPC code to execute it. A trap instruction is installed at the end of the inferior RPC code to notify Paradyn that it can then resume the application. Inferior RPC is similar to the "call" feature of a debugger [4], and is a complement to Paradyn's dynamic instrumentation.

For a threaded program, we need to execute inferior RPC code for a particular thread (e.g., when a thread requires starting a timer). But Paradyn has no control over which thread will be executing in the application when performing the inferior RPC. Our solution is to allow any thread to execute the code *on behalf of* the particular thread that needs to run the inferior RPC. We achieve this by passing the ID of the thread to the inferior RPC and have the inferior RPC code look up the correct entry in the Thread Table.

When we perform an inferior RPC, the thread that is executing may be the one for which we want to perform the asynchronous operation. This asynchrony could cause problems similar to instrumenting thread context switches. To avoid this problem, we created safe inferior RPC, in which Paradyn posts inferior RPCs in a shared memory segment accessible by both Paradyn and the application; the base-trampolines check this segment for pending RPC's and execute them at a safe time. This ensures that inferior RPC will not interleave with other instrumentation.

3.5 Refining Paradyn's Resource Hierarchy

Paradyn decouples performance metrics from program components. A user can request performance data such as CPU time spent by a particular procedure by selecting the appropriate procedure from the resource hierarchy and the performance metric *CPU Time*. We extend the resource hierarchies of Paradyn to include threads and a number of thread-specific synchronization objects (condition variables, mutex locks, and read/write locks). The left side of Figure 5 shows the code hierarchy that lists the procedures in the shared object `libjava_g.so`. The middle part shows the process hierarchy where the process `java_g{12192_chocolate}` has several threads. The right side shows the resource hierarchy for the synchronization objects.

Figure 6 shows an example of profiling synchronization waiting time on conditional variables for two individual threads. The top curve shows the conditional waiting time in the thread `thr_1` and the bottom curve shows the conditional waiting time in the thread `thr_7`.

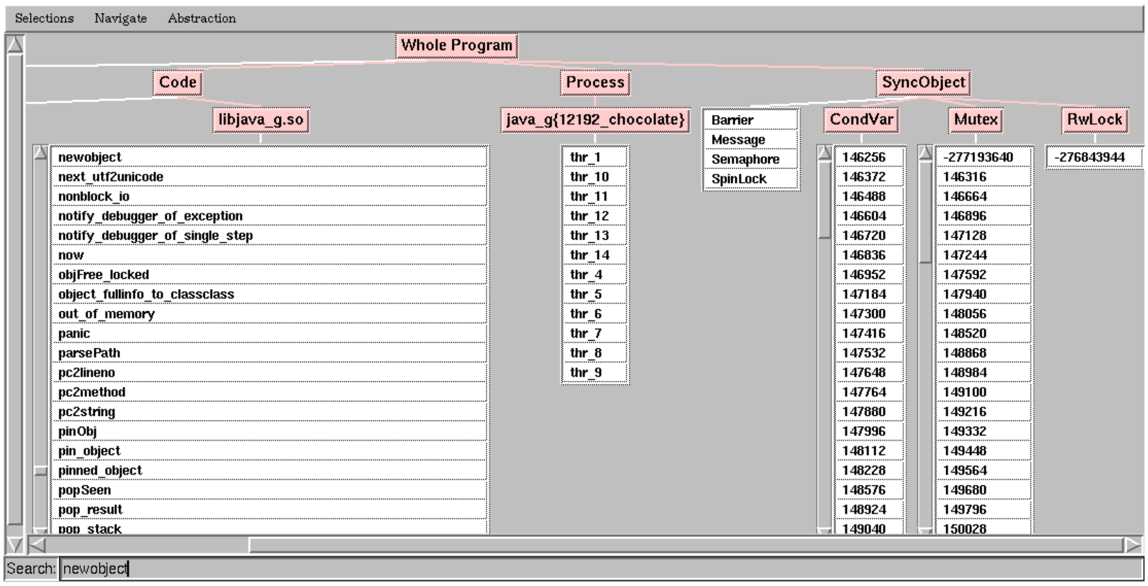

Figure 5: Resource Hierarchy

Showing resource hierarchies for code, process and synchronization objects.

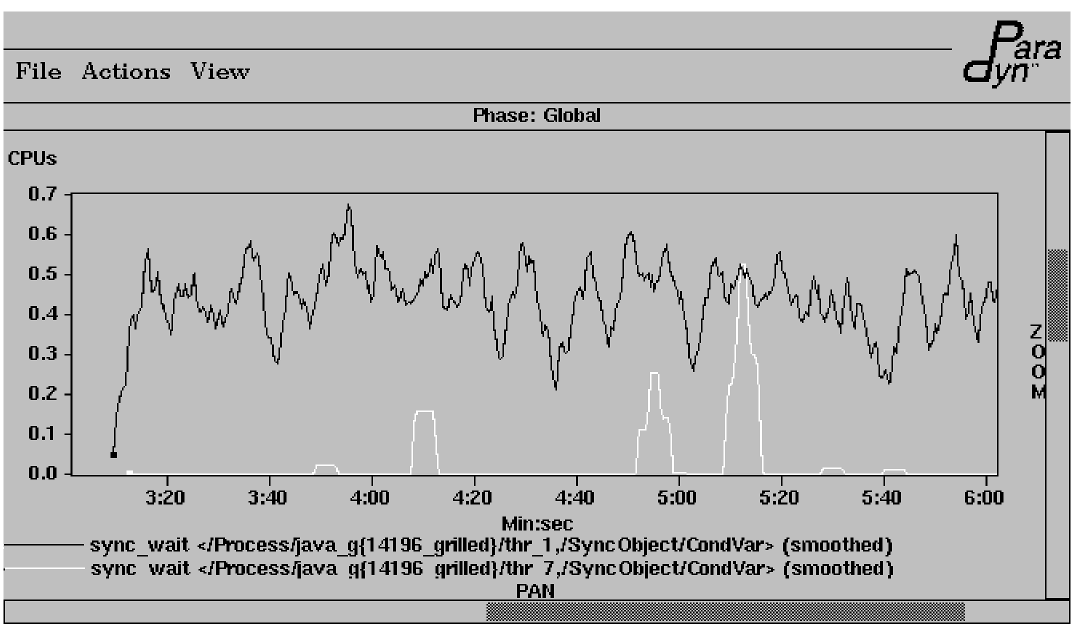

Figure 6: Fine-Grained Performance Data

Dividing synchronization waiting time to conditional variable and individual threads.

4 INSTRUMENTATION OVERHEAD

This section describes the cost of instrumenting threaded code through a few simple benchmarks and compares them with non-threaded Paradyn. Table 2 presents the cost of basic instrumentation primitives, including the cost to execute the base-trampoline, increment a counter, and start and stop a CPU timer. These costs are presented for the threaded and non-threaded Paradyn instrumentations on two different systems: an UltraSPARC II with one 250 MHz processor, and an Enterprise 5000s with twelve 167-MHz processors.

Machine	Base Trampoline		Counter		Start Timer		Stop Timer	
	Non-threaded	Threaded	Non-threaded	Threaded	Non-threaded	Threaded	Non-threaded	Threaded
UltraSPARC II Uniprocessor	124ns	573ns (+362%)	27ns	39ns (+44%)	1.9µs	3.1µs (+63%)	2.0µs	2.9µs (+45%)
Enterprise 5000s	185ns	773ns (+318%)	40ns	55ns (+38%)	2.8µs	4.6µs (+64%)	3.0µs	4.4µs (+47%)

Table 2: Micro Benchmarks

As shown in Table 2, the cost of base-trampoline for threaded Paradyn is about 5 times of that of the non-threaded version. The extra costs are from code added to check for inferior RPCs and to calculate the column address in the Thread Table. Counter primitives for the new version are 40% more expensive than the non-threaded Paradyn, and timer code is about 60% more expensive, mainly because the new version has to go through a level of indirection.

Instrumentation	Non-threaded Paradyn (Sequential Version)		Threaded Paradyn (Threaded Version)	
	UltraSPARC II 1 processor	Enterprise 5000s	UltraSPARC II 1 processor	Enterprise 5000s 4 processors
No Instrumentation	64.6s	96.3s	65.0s	24.4s
CPU Time (Inclusive) Whole Program	66.1s (+2%)	97.5s (+1%)	67.3s (+4%)	25.2s (+3%)
Procedure Call Frequency Function `innerp`	67.5s (+4%)	97.8s (+2%)	68.3s (+5%)	25.8s (+6%)
CPU Time (inclusive) Function `innerp`	67.9s (+5%)	100s (+4%)	71.9 s (11%)	27.3s (+12%)

Table 3: Matrix Multiply

To get a feeling for the overall cost of the new instrumentation, we instrumented a simple multithreaded application (matrix multiply with 150 lines of C code) and compared it with the cost of instrumenting a sequential version of the same algorithm by the non-threaded Paradyn. Table 3 shows elapsed times of the two versions repeatedly multiplying two 500x500 matrices of floating point numbers. In Table 3, we measure CPU time (inclusive) for the whole program, procedure call frequency and CPU time (inclusive) for the function `innerp`. The procedure call frequency of `innerp` is about 3,500 calls/second on the uniprocessor, and about 10,000 calls/second for the multithreaded version on the multiprocessor. Note that the overhead for the threaded instrumentation is about 2 to 3 times of that for the non-threaded instrumentation. Instrumentation cost is proportional to event frequency. In this example, we instrumented the most frequently called procedure as a stress test.

5 DISCUSSION

In the previous sections, we have described our techniques to dynamically instrument threaded programs. There are several things we can do to make our tool more usable.

First, we can improve the naming of threads and synchronization objects in the resource hierarchy to better relate them to source level constructs. Since we operate directly on binary programs, this information must be extracted with our runtime instrumentation. For threads, we can label them with the start function, thread ID, or an application-provided string name that is provided by some thread packages. Associating source-level information with synchronization objects is more challenging, since most synchronization object can be created without explicitly calling a creation function. However, if a synchronization object is created by calling a creation routine, we can associate it with the line number and source file name of the calling function.

Second, we have focused our attention on the correctness of instrumentation so far. Instrumentation overhead for threaded code is still more expensive than that for non-threaded one. There are several places we can make our instrumentation code more efficient. For example, we can collapse the logic of our performance timers with that of the per-thread virtual timer to make the timer code more efficient. We can also use hardware timers for wall time-based metrics.

Last, most processors provide hardware counters and timers for performance metrics such as cache misses and branch mispredictions. Incorporating these measurements into our tool will allow us to provide the users with more useful information.

6 PRELIMINARY EXPERIENCE

Using the new threaded instrumentation, we studied the performance of the Sun Java Virtual Machine version 1.1.6 interpreting the AppletViewer driven by a game applet called Tetris (from Java Boutique) on a UltraSPARC uniprocessor machine running Solaris2.6. The Tetris applet[2] has approximately 1000 lines of Java code. The Tetris applet and AppletViewer adding together include about 280 Java classes.

Figure 7 shows CPU time for the individual threads. Of the 14 threads, 3 threads accounted for most of the time; Thread `thr_7` was the largest, followed by threads `thr_13` and `thr_14`. The "invoke" functions (e.g., `invokeJavaMethod` and `invokeNativeMethod`) are common bottlenecks in Java. Profiling these functions showed that the function `invokeNativeMethod` took about 40% of the total CPU time (see Figure 8). A more detailed look reveals that `invokeNativeMethod` is mostly called by the thread `thr_7`, accounting for about 50% of the time spent in `thr_7` (see Figure 9). A close examination of `invokeNativeMethod` in `thr_7` showed that most

2. The applet has delays inserted to artificially slow its visual behavior. To obtain a more demanding load on the virtual machine, we removed these delays.

of its time is due to a call to `sun_awt_motif_X11Graphics_drawLine` (see Figure 10). We were able to eliminate some redundant code shared by the monitor enter and exit code and cut the number of calls to the function `sysThreadSelf` (which returns information about the currently running Java thread) by about 75% in thread `thr_7`. Figures 11 and 12 show the number of calls to `sysThreadSelf` before and after redundant code elimination. As a result, we improved the performance of the function `sun_awt_motif_X11Graphics_drawLine` by about 22% and elapsed time of the JVM by about 7% (see Table 4). In this performance study, we used the ability to select performance data for an individual function or individual thread, and for an individual function *only for* an individual thread (this feature appears unique to our tool).

	sun_awt_motif_X11Graphics_drawLine	Elapsed Time
Original	29.8 µs	20.3s
Optimized	23.2 µs (-22%)	18.85s (-7%)

Table 4: Performance Improvements

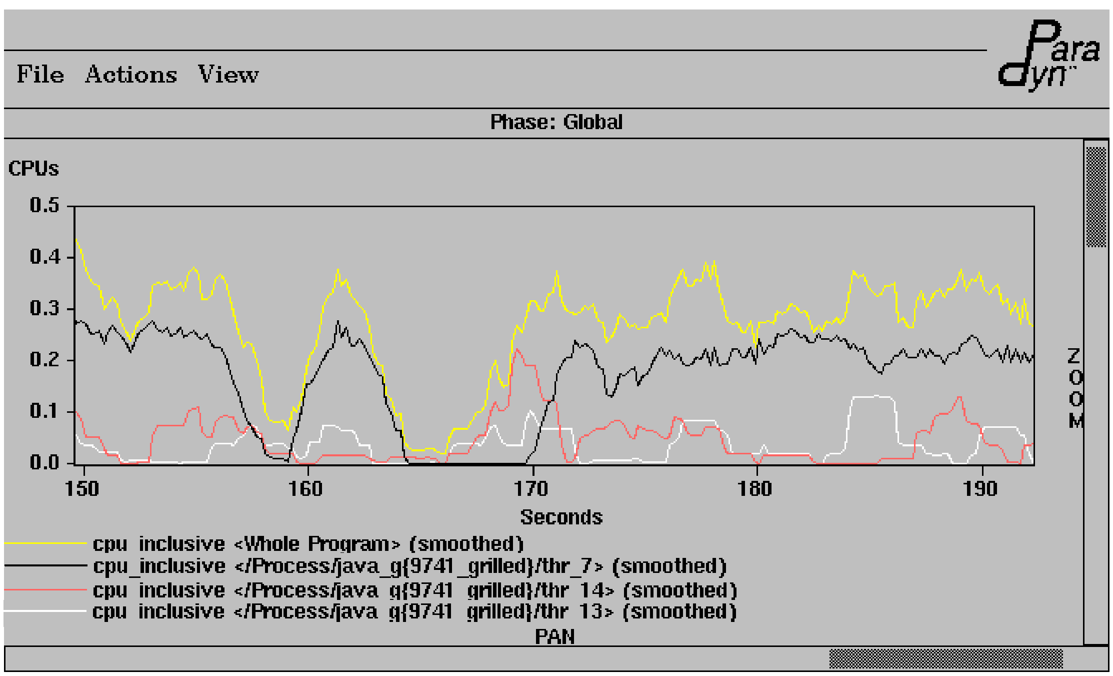

Figure 7: CPU Time Measured on a Per-Thread Basis

Of the total 14 threads, thr_7, thr_13 and thr_14 accounted for most of the CPU time. The top curve shows the CPU time spent by the whole program, followed by curves showing time spent by threads thr_7, thr_13 and thr_14.

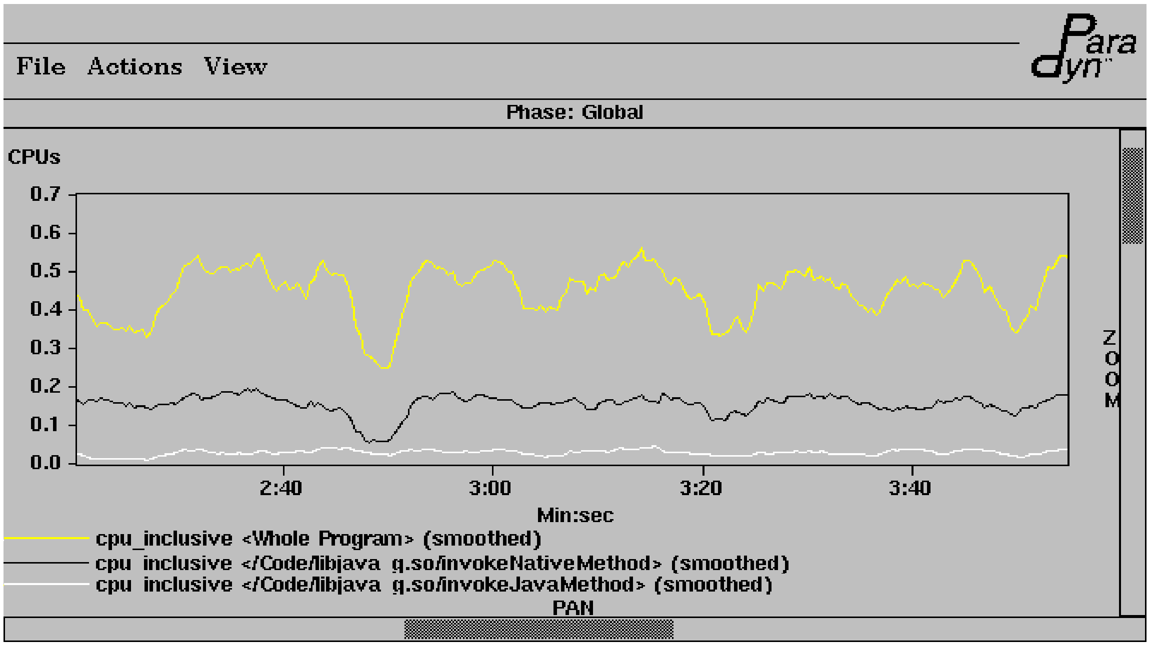

Figure 8: invokeNativeMethod takes about 40% of total CPU time

The top curve shows CPU time spent by the whole program, followed by the curve showing CPU time spent on calling invokeNativeMethod.

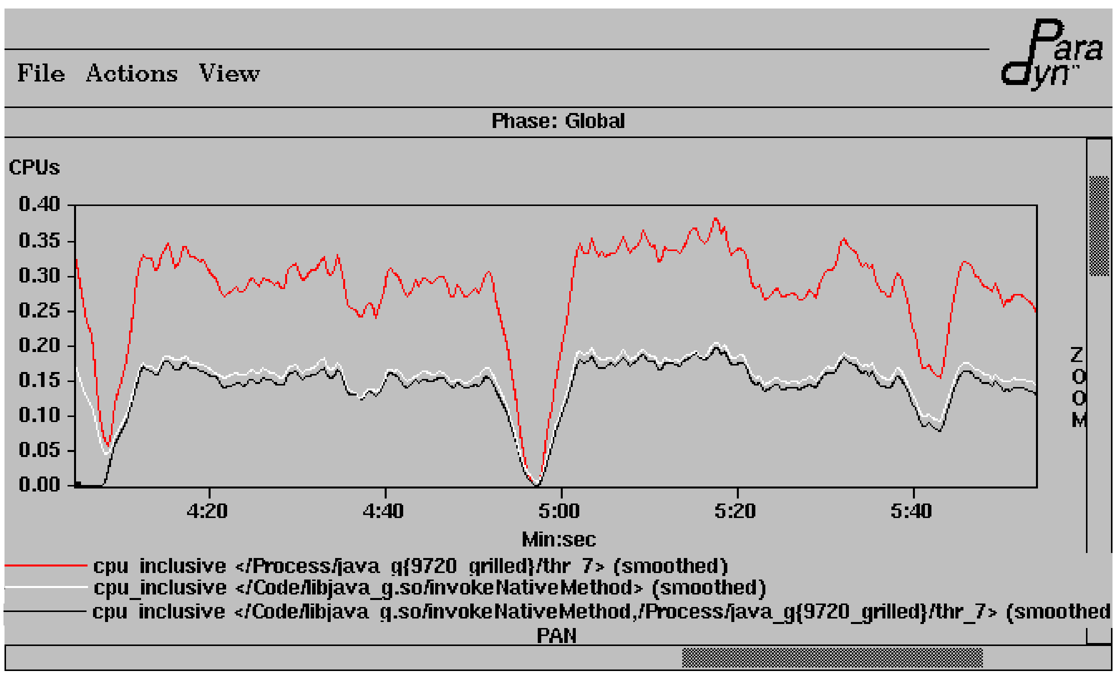

Figure 9: invokeNativeMethod was mostly called by thr_7

The top curve shows the total CPU time spent by the thread thr_7, the middle curve shows the CPU time spent on calling invokeNativeMethod by the whole program, and the bottom curve shows the CPU time spent by thr_7 calling invokeNativeMethod.

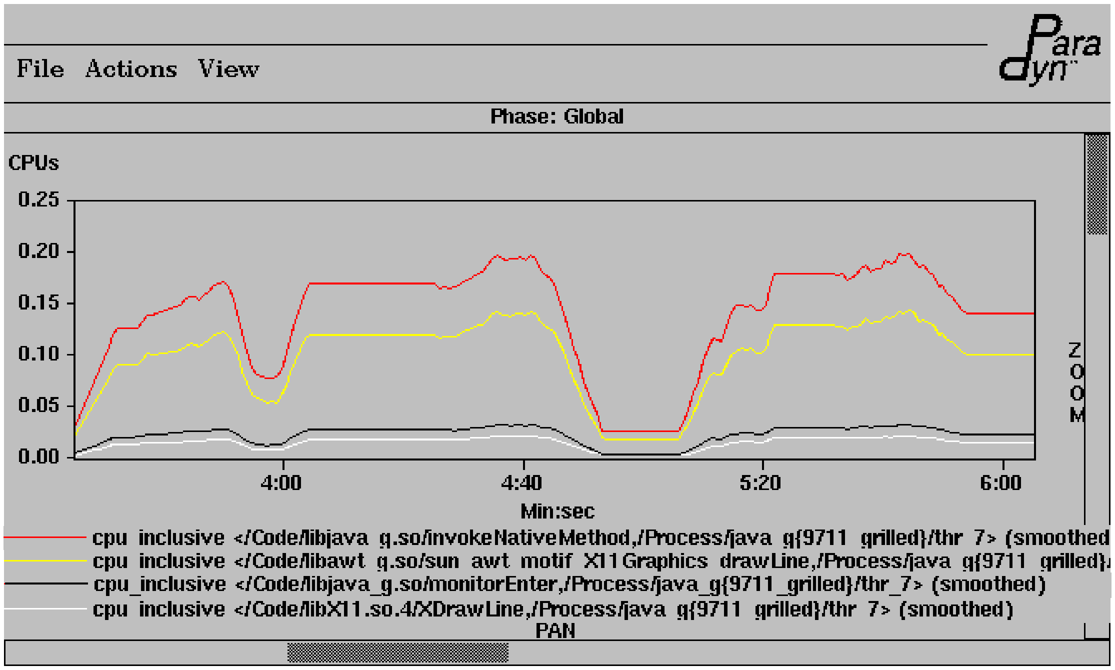

Figure 10: Subdivide invokeNativeMethod

Most of the CPU time spent by invokeNativeMethod in thr_7 is on calling sun_awt_motif_X11Graphics_drawLine, and only a small percentage of which is spent on the "real work" XDrawLine.

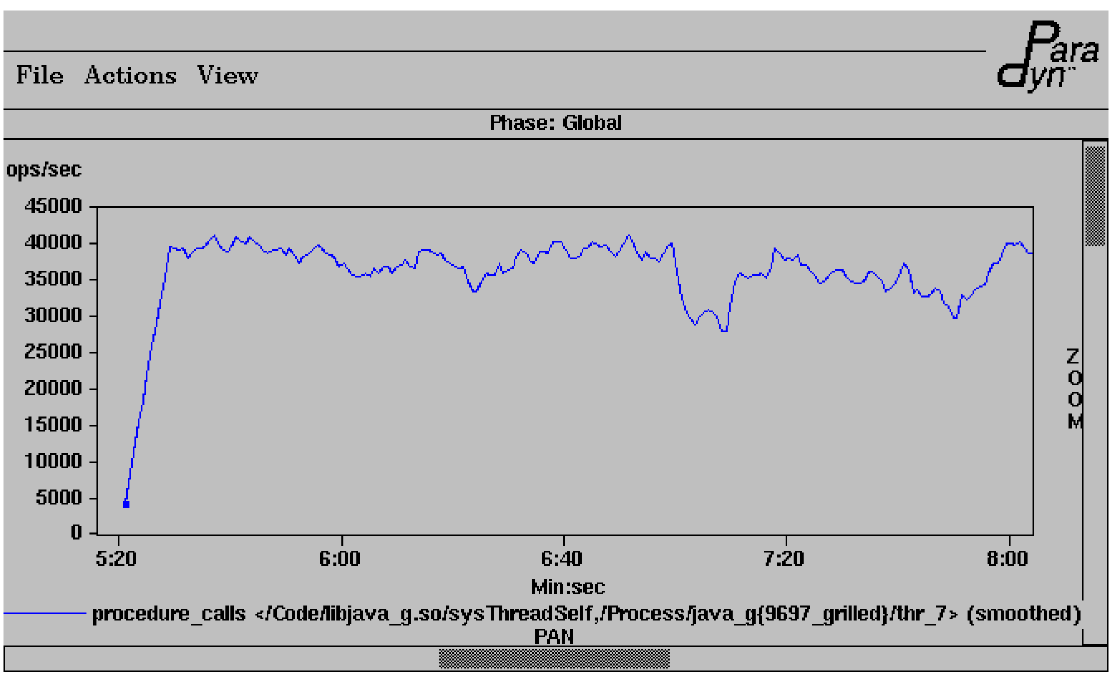

Figure 11: Calls to invokeNativeMethod and sysThreadSelf

For each call to invokeNativeMethod, there is approximately 4 calls to sysThreadSelf. The number of calls to monitorEnter, monitorExit and XDrawLine are the same as the number of calls to invokeNativeMethod.

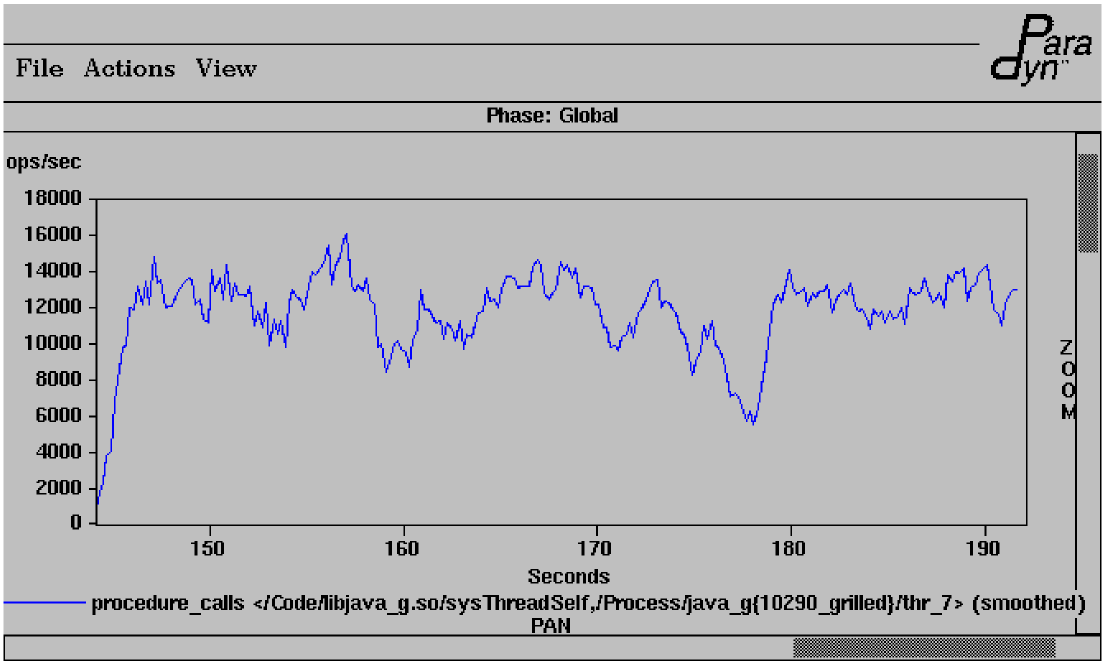

Figure 12: Calls sysThreadSelf After Redundant Code Elimination

After eliminating redundant code, calls to sysThreadSelf is cut significantly, and number of calls to invoke-NativeMethod increased slightly.

7 Related Work

While many tools have been developed to monitor and analyze the performance of parallel programs, few tools have been built for multithreaded programs. The Sun WorkShop™ Thread Event Analyzer [8], Socrates [7], and Tmon [3] are all trace-based tools that are able to divide performance data to threads. Both the Event Analyzer and Socrates save trace data for post-mortem analysis and visualization. Socrates stores the trace data in a database to be used by other analysis and visualization tools. The Sun Event Analyzer measures file and socket I/O, memory allocations, string usage and synchronization, and visualizes them in graph, table or timeline displays. Both Socrates and the Event Analyzer allow a user to select events of interest through graphical user-interfaces to control instrumentation overhead. In the Event Analyzer, instrumentation is accomplished by shared library interposition, and instrumentation in Socrates is through changing a program's source code.

Tmon is a performance tool for multithreaded programs developed at Princeton University. Tmon implements performance metrics that are at a higher-level than those implemented by the Event Analyzer. The metrics Tmon implements include thread waiting time (building thread waiting graph), and semi-busy-waiting time. Tmon supports on-line analysis and visualization, and it uses a client-server architecture where the server side of Tmon processes traces sent from the client side via a socket. Tracing and sending traces via a socket can incur intrusion as large as 3.3 times slow down [3]. The same as Socrates, Tmon captures thread synchronization events by changing the source code of a lightweight user-level thread package.

Gthread [9] is a trace-based Pthread Visualization Package developed for the KSR parallel machines to debug multithreaded programs. Gthread involves a set of animation views that depict the individual threads and their movement through the functions of the program, and shows other program features such as barriers and mutexes. Gthread adds instrumentation to a program by using macros to replace the Pthread calls to add tracing capability. It writes the tracing data to a file that is subsequently used by the visualizer to drive the animation.

Our tool differs from other tools in that we use runtime dynamic instrumentation, without tracing. Dynamic instrumentation allow our tool to instrument unmodified binary and running programs, and allow us to control instrumentation overhead dynamically.

8 CONCLUSION

This paper describes our design and initial experience with non-trace based performance instrumentation techniques for threaded programs. Our goal is to provide detailed performance data while maintaining control of instrumentation cost. Our main contributions are the SICMD design to divide performance data to individual threads, and techniques to handle asynchrony and contention in instrumenting multithreaded programs.

With SICMD, all threads share the same instrumentation but each has its own private copy of performance data. This approach simplifies instrumentation code and reduces the use of locks. Instrumenting multithreaded programs faces new challenges such as dealing with contention and asynchrony. We introduced thread-conscious lock to avoid a scenario where an instrumentation can deadlock a thread, and per-thread virtual timers to reduce the number of calls to expensive timer routines at thread context switches. Per-thread virtual timer also reduces the chance of interleaved instrumentation accessing the same performance data. Along with other techniques such as low-contention data structures and caching, instrumentation cost for a threaded program is comparable to that for a non-threaded program.

Our initial implementation is on SPARC Solaris2.x including multiprocessor SUN UltraSPARC Enterprise machines. We show how our new technique help us to speed up a Java native method by 22% and overall execution time of the JVM interpreting the AppletViewer driven by a game applet by 7%.

Acknowledgments

We thank Tia Newhall for helping with Java and JVM, Matt Cheyney and Ari Tamches for helping with Paradyn, Tia Newhall, Ari Tamches and Brian Wylie for their comments on this manuscript, Dan Nash for suggesting the per-thread virtual timer, and other members of the Paradyn group for many helpful discussions.

REFERENCES

[1] B. P. Miller, M.D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer 28*, 11, November 1995.

[2] J.K. Hollingsworth, B.P. Miller, M.J.R. Gonçalves, O. Naim, Z. Xu and L. Zheng. MDL: A Language and Compiler for Dynamic Program *Instrumentation. International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, CA. Nov., 1997.

[3] M. Ji, E. W. Felton and K. Li. Performance Measurements for Multithreaded Programs. *ACM SIGMETRICS/Performance*, 1998.

[4] M. Loukides and A. Oram. *Programming with GNU Software.* O'Reilly & Associates, Inc. Jan. 1997.

[5] J. Ousterhout. Why Threads Are A Bad Idea (for most purposes). *Invited talk at the 1996 USENIX Conference*, 1996. Also available at http://www.scriptics.com/people/john.ousterhout/threads.2up.ps. (Sept. 28, 1995).

[6] U. Vahalla. *UNIX Internals: The New Frontiers*. Prentice Hall.1996. pp.48-80.

[7] A. Voss. Instrumentation and Measurement of Multithreaded Applications. *Thesis.* Institut fuer Mathematische Maschinen und Datenverarbeitung, Universitaet Erlangen-Nuernberg. Jan. 1997.

[8] P.-T. Wu & P. Narayan. Multithreaded Performance Analysis with Sun WorkShop™ Thread Event Analyzer. *Technical White Paper.* April 1998, Revision 03.

[9] Q. A. Zhao and J. T. Stasko. Visualizing the Execution of Threads-based Parallel Programs. *Technical Report GIT-GVU-95-01*, Graphics, Visualization and Usability Center, Georgia Institute of Technology, Atlanta, GA. January 1995.