

Ok For MS research Req'd
David Yamblich
11/4/02
Laidlaw

TIV: Thread Interaction Visualizer

Kevin Forbes Audleman
committee

David Laidlaw, Steven Reiss, Shriram Krishnamurthi

August 30, 2002

1 Abstract

This paper describes Thread Interaction Viewer (TIV), a monitoring tool for examining the behavior of large multithreaded programs. A number of tools have tackled the problem of multithreaded program understanding through the visualization of thread scheduling information. This information allows the programmer to see when threads are running, waiting to run, or unable to run due to blocking on locking primitives. This approach is good for finding inefficiencies in a threaded application, as they are often caused by excessive or unexpected blocking. However these tools do little to describe why this blocking is occurring. In modern software this can be one of many reasons, and distinguishing between them is necessary to understand the nature of the problems. TIV extends upon existing work by using a number of visualizations to describe the type and quality of blocking. TIV represents data in a way that provides an efficient means for examining large and long running programs. This is facilitated in part by a flexible visualization framework that uses levels of detail to modify the visualization for different numbers of threads. We have found TIV to be good at identifying problems that are the result of erroneous synchronization on shared mutexes, errors in thread scheduling, and other situations where threads are interacting improperly. It can also be used to identify I/O behavior. We demonstrate TIV on stored traces of Java programs. However it can easily be extended to work with other languages and thread libraries.

2 Introduction

Multithreading is a powerful technique to exploit parallelism on multiprocessors and speed up execution on uniprocessors by interleaving multiple paths of execution. However, the interaction of multiple threads introduces a new set of problems that can negatively impact program performance or cause erroneous execution.

Problems stem from the complex manner in which threads interact. Threads execute in the same address space and share common memory. Multiple threads can execute in the same code simultaneously, using and modifying the same variables and objects. This can cause serious problems if there are sections of code where it is only safe for one thread to execute at a time. For instance, if two or more threads attempt to write to the same variable simultaneously, the resulting value can be undefined. To combat this problem, locking primitives are used to synchronize access to critical regions. Segments of code are serialized by surrounded them by a lock; threads sleep in a wait queue on the lock until they are allowed access.

The proper use of synchronization is crucial to program behavior, but is difficult to get right. Small errors in logic can have a large impact on performance and correctness. Excessive blocking can outweigh the benefits of threads running in parallel, slowing the program down to a crawl. Small flaws in the synchronization logic can cause threads to behave incorrectly, even deadlocking in the worst case.

Specialized tools are necessary to understand these problems and develop robust multithreaded programs. An examination thread scheduling information is a key component in identifying thread behavior. However, traditional tools do not do a good job of displaying this information. As a result, developing tools to fill this niche has been an active area of research. We discuss several such tools in our related work section.

As programs get more sophisticated and the use of synchronization becomes more complex, understanding their behavior becomes even more difficult. For instance, in a socket based message server, there will typically be several threads listening for messages, sleeping on a lock while they are waiting. In this case, large amounts

of sleeping describes *correct* behavior; indeed, if none of the listener threads were sleeping, there would not be any threads ready to accept messages. Furthermore, the exact nature of why threads are blocking can be due to one of many reasons. A group of threads swapping control of a lock, and one thread holding a lock several others are waiting for both are very different situations. Identifying not only the location, but the nature of synchronization is important to understand multithreaded programs. There is a need for new tools that help in understanding this new level of complexity.

We have developed TIV, a visualization tool that makes it possible to identify these distinctions. TIV has two main features. First, it uses a combination of two views to present the data. The combination provides a way of identifying why threads are behaving the way they are. The *timeline view* provides a play by play recall of the execution, and the *interaction view* provides details as to why things are happening.

Second, it employs levels of detail to handle programs of large scale. As the number of threads on the screen increases, the visualization changes to a higher level of detail, displaying less information but maintaining a readable visualization. Close up views are good for few threads and can be used to look at data in full detail. Farther views pair down the information to make a more readable display and focus on higher level concepts. Levels of detail make the effective number of threads TIV can visualize range from one to hundreds.

The rest of this paper is divided into the following sections. Section 2 discusses related work. Section 3 describes the technologies we use to implement TIV and the system foundations. Section 4 discusses the data parsing we do to extract behavior information. Section 5 describes our visualization in detail. Section 6 discusses our results, and section 7 future work.

3 Related Work

Prior work in visualization of concurrent programs began with compiler-driven (FORTRAN) parallel programs. Systems such as The D Editor [7] and others [8] provide a visual way of locating sections of code containing sequentialized code or expensive communication.

More relevant to our work have been systems developed to specifically examine threads-based parallel programs. Early work in this area provided a way to examine multiple paths of execution simultaneously and the communication between them [1] [2] [3]. Jinsight [5] by IBM is a tool that combines these features with several views that display thread activity, object usage and creation, and garbage collection. However, none of these tools provide information about thread scheduling, which is the cornerstone of TIV.

Tools designed to illustrate thread scheduling information have come about more recently. Thread-Mon [11] is one such tool that provides a way of visualizing thread activity as well as the way they map to both LPWs and CPUs. The Java Instrumentation Suite [4] provides a way to examine a detailed time analysis of thread scheduling activity done by the Java Virtual Machine. The developers of the JX language, a Java-based operating system, created a set of tools to let them visualize the performance of the micro-kernel and Java components [6], including several thread scheduling views. We will demonstrate how TIV reproduces and expands on the work of these tools. It does so through a unique set of visualizations that makes answering the question of *why* synchronization is occurring possible, and increases the maximum size of programs that can be examined.

4 TIV Basics

TIV visualizes the execution of multithreaded programs, gathering its data from stored traces of Java programs. TIV could easily be extended to work with data from other sources, however. Appendix A describes the API TIV defines to gather program information. It is general enough that it should be applicable to most languages and thread implementations. Any source that can provide calls to these routines can be used with TIV.

4.1 Data acquisition

We chose to work with traces of Java programs. This was due in part to us having access to a suite of tools for gathering traces. However, Java is also an excellent language to work with. It is widely popular and has a threads package built in to the language, which means there's a large code base to work with. Java also has need for debugging tools.

Traces are gathered using the `javatrace`[1] utility. A program is invoked with `javatrace`, which executes like normal with the only noticeable difference being a slowdown due to the tracing. `Javatrace` produces a trace file that describes the entire execution of the program. This trace is then input into `TIV`, which parses it with the `tfile`[2] library

4.2 Data Parsing

`TIV` extracts behavioral information about the program from the trace. It gathers both the states of execution and details about blocking behavior of the threads.

4.2.1 States

We collect summary statistics of the actions a thread takes over time and report them as states. These states reflect the activity of each thread in the application through a set of predefined states that are representative of the parallel execution. The states are reported over a user controlled length of time, which in all the examples in this paper is one second.

Running is measured per thread, and describes how much time the thread spent running on a processor. On a single processor, the summation of this number across all threads cannot exceed 100 percent. On multiple processors it can be higher, although it still has a bound.

Runnable describes the time a thread spends waiting for a processor. When there are more threads than processors, each thread will spend a fair amount of time simply waiting for its turn to run.

Suspended is the time a thread is not in consideration for execution; in essence when it is put to sleep. This can occur when a thread calls functions like `sleep`, `wait`, or `join` on itself. Or it can occur when another thread calls `suspend` on it.

Blocking is the time a thread spends suspended on the wait queue of a monitor.

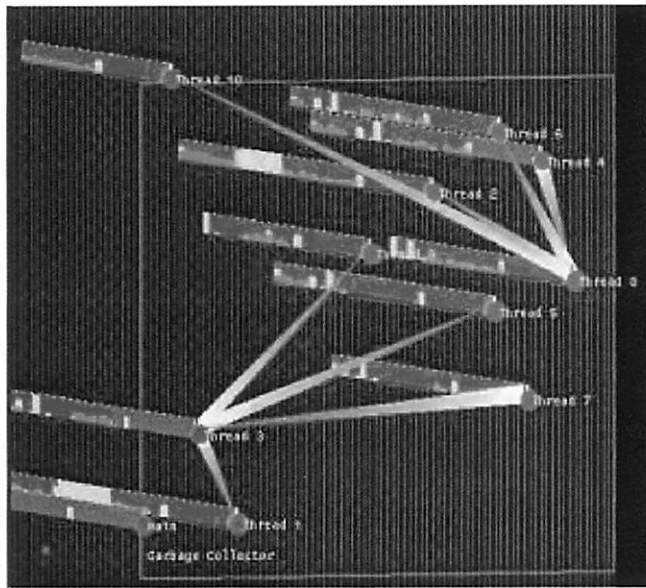
I/O Waiting is the amount of time a thread spends blocking on calls to I/O. This includes all call to memory and sockets.

4.2.2 Blocking details

The time a thread spends in the **blocking** state is further divided into how much time it spent waiting on other threads. For instance, if `Thread 2` and `Thread 3` each hold a lock for half the time `Thread 1` is waiting on it, `Thread 1` reports blocking on `Thread 2` 50% of the time and `Thread 3` the other 50%.

4.2.3 Time steps

The user can control the length of time states are reported over. We have found one second to be a good amount of time for an initial examination of a program. However different time lengths provide `TIV` with the flexibility to view the data in different ways. Decreasing this time to a fraction of a second will give a much more detailed breakdown of states. If the exact timing of thread blocking is important, this will help. On the other hand, a longer time can smooth out excessive detail when a general overview is required. It also allows more execution time to be visualized at once.



Legend

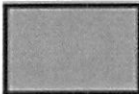
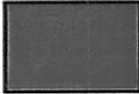



	Suspended
	Blocking
	I/O Waiting
	Runnable
	Running

Figure 1: 3-D view of TIV. The buttons in the lower left control the playback. The mouse is used to rotate the viewpoint around. Each line represents a thread of execution in the program. The colors on them represent the states. The blue lines between threads represent mutual blocking.

4.3 Data Examination

TIV starts when the user loads a trace of a Java program. It sets up a 3-D visualization window that shows the data, then begins the playback. TIV plays back the program like a VCR at about the same speed, with a slowdown around a factor of four due to the trace file parsing. The user can use the pause button to halt the playback at any time, and the play button to resume it.

5 Visualization

TIV creates a visualization of the data to aid in its examination. Our tool graphically represents the threads, their states, and their interactions in a three dimensional space (figure 1).

Each bar in three space represents one of the threads in the program. The z-axis represents time (into the screen), with the near X-Y plane being the most recent time step. The x and y axes are used to spatially place the threads. The colors and textures on each thread represent their states, while the lines between them represent their blocking.

The user can move the camera to any position in the space to find the optimal viewing angle. Zooming in and out allows either a close look at a few threads, or an examination of all of them at once.

There are two main traits the visualization has that make it interesting. The first is its ability to not only locate where and when problems in synchronization are occurring, but answer *why*. It does this by using two viewpoints together, the **timeline view** and the **interaction view** (figure 2).

The second is TIV's ability to visualize programs of a large scale. It does this by using a scalable data representation, providing the ability to spatially cluster elements, and implementing a level of detail system. These traits are described in detail below.

5.1 Timeline and Interaction Views

The timeline view and interaction view display the two sets of data TIV gathers: state information and blocking details. The position of each thread in the two views lines up, so both sets of information for each

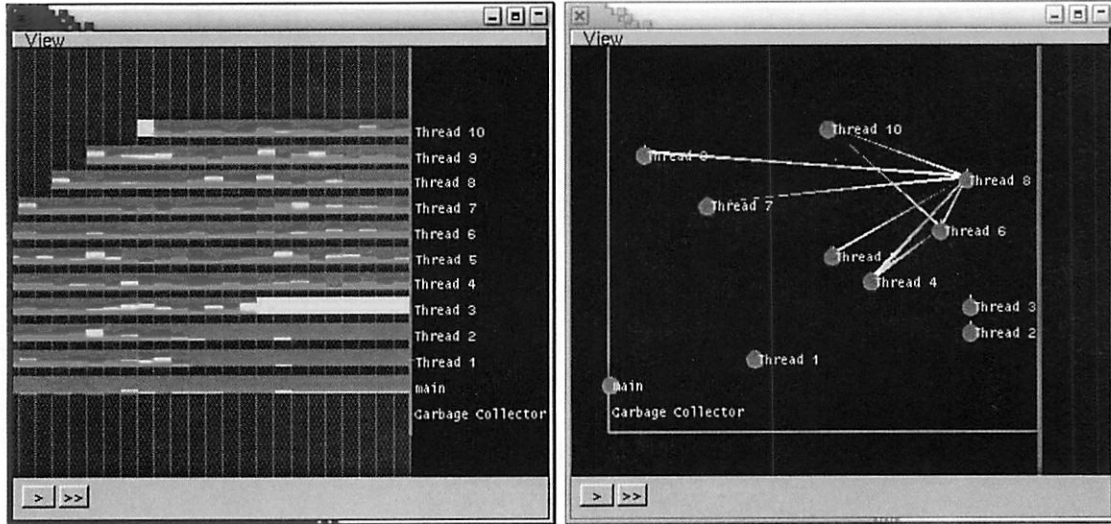


Figure 2: Left: The timeline view is used to see what states threads are in. Right: The interaction view is used to see how threads are working together.

thread can be easily distinguished.

5.1.1 Timeline View

This view is used to examine thread states. Each state is represented by a colored bar (see the legend in figure 1, with the vertical height representing the percentage time spent in that state).

The first four categories, running, runnable, suspended, and blocking describe thread scheduling, and are the states that have been represented in previous work. I/O Waiting, however, is in a different category, and TIV is novel in including it with thread scheduling information. Doing so lets TIV identify additional program events and behavior. For instance, TIV can identify areas of code that are running slowly due to large amounts of disk access. Additionally, TIV can distinguish between different types of blocking, such as blocking on a socket versus blocking on a monitor. While these are technically the same thing, they are very different conceptually. We will demonstrate this later on a message server, where this distinction is critical.

5.1.2 Interaction View

This view shows the nature of thread synchronization. The vertical position of a thread is the same as in the timeline view, and the horizontal axis is used to space threads out.

Connections between threads denotes blocking and are represented by triangles. Thickness of the connecting triangle indicates the degree of connectivity. The narrow end represents the originating end, and the wide end the thread it is blocking on.

A slider can be used to control how many timesteps worth of blocking behavior to show. Showing multiple timesteps is beneficial because it allows us to identify trends. When we break up the execution into time segments, threads that are working together may or may not block on each other at any given time step. If we only showed one time step, it would result in a rapidly changing picture as connections appeared and disappeared. Multiple timesteps smooths this out and shows how threads are interacting over time. On the other hand, looking at a single timestep can be useful if a detailed understanding of timing is required.

This view allows TIV to differentiate between different patterns of blocking that would otherwise appear identical. Figure 3 shows three possible patterns. A single thread could be holding a lock while other threads wait for access. A group of threads could be sharing a lock equally. Or groups of threads could be blocking on entirely separate locks.

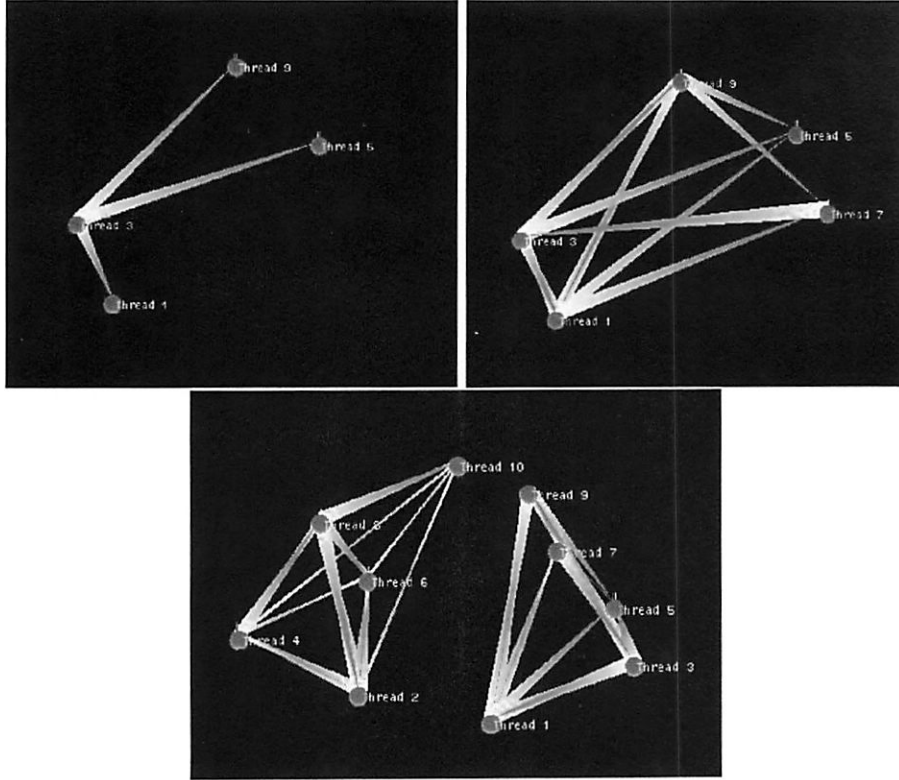


Figure 3: *Top left: Many to one.* Three threads are waiting while one thread dominates a lock. *Top right: Many to Many.* Several threads swap control of a lock. *Bottom: Multiple groups.* Threads are grouping on different locks.

5.2 Visualizing large scale programs

TIV is designed to visualize large scale programs. This means programs with several hundred active threads and a runtime of hours or days. The following features make TIV work at these scales.

5.2.1 Moving and Grouping

TIV allows the user to position threads anywhere in the 3D space. This improves the readability of the display by allowing the user to cluster logically related groups of threads. Clustering also allows the user to move uninteresting threads out of the way.

5.2.2 Levels of Detail

The visualization discussed so far has the problem that it breaks down when too many threads are on the screen. Beyond 10 - 20 threads, each thread gets so small that the states cannot be made out. At the same time, it becomes less important to see the exact breakdown of each thread, as the relevant behaviors at this scale occur at the group level. A modification of the original visualization that shows less detail provides a solution to both of these problems. We have developed a level of detail system to handle this problem. The system works by eliminating detail about each thread as the number of threads on screen increases. The user can zoom in and out to view specific threads at different levels of detail.

We have defined three levels: near, mid, and far. Near is designed for up to ten threads, mid up to 60, and far up to 200. Each level emphasises different data by defining a new set of states, as can be seen in figure 4. The higher levels of detail make new states by merging states from lower ones. The levels have the following traits.

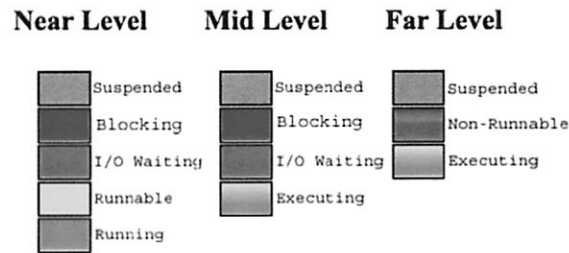


Figure 4: Levels of detail

- The mid level replaces **running** and **runnable** with the single state **executing**. Running and runnable are used to identify how much time a thread spends on a processor versus how much time it spends waiting. This can highlight problems such as a thread being starved as higher priority threads hog the processor. However, in large systems making this distinction is often unimportant. Programmers will leave the threads package to do the scheduling, and assume it does a good enough job (which it usually does). In this case, abstracting away this information allows the programmer to focus on what is relevant.
- The far level replaces **blocking** and **waiting I/O** with the single category **non-runnable**. This describes the time a thread is not executing because it is forced to wait for some resource. This level is designed for up to 200 threads, where screen space is at a real premium.

6 Application analysis with TIV

In this section, we highlight some conclusions drawn from our initial experimentation with JIS. The main idea is to show the usefulness of the tool in analyzing the behavior of a threaded application, understanding the behavior of the underlying thread library, and identifying a variety of higher level problems that occur through threads interacting. We ran TIV on a number of programs, including a number of samples we wrote and a real program written by Steve Reiss.

6.1 Garbage collector problem

The following problem is attributed to Sevitsky, a developer at Sun. His team had a piece of code that was running extremely slowly for some unexplainable reason. They analyzed it with Jinsight, and discovered that the garbage collector was unusually active. With this information they were able to track the problem down to someone including a call to invoke the garbage collector (`System.gc()`) in the finalize method of a class. As this class was being allocated and deallocated repeatedly, the garbage collector was running almost constantly.

Figure 5 shows TIV's timeline view on two versions of a program that simply allocates class objects repeatedly. On the left, a call to `System.gc()` is included in the object's finalize method. As can be seen, the garbage collector is highly active. On the right the call is taken out, and the garbage collector is back to normal.

6.2 Priority program

The following program uses `java.lang.Thread.setPriority` to set the priority of one thread higher than the others. The expected outcome is that the thread with the high priority will get more time in the running state than the others. However TIV provides a way of examining this data. Figure 6 shows the timeline view of a run of the program. Oddly, all of the threads are getting approximately the same amount of time

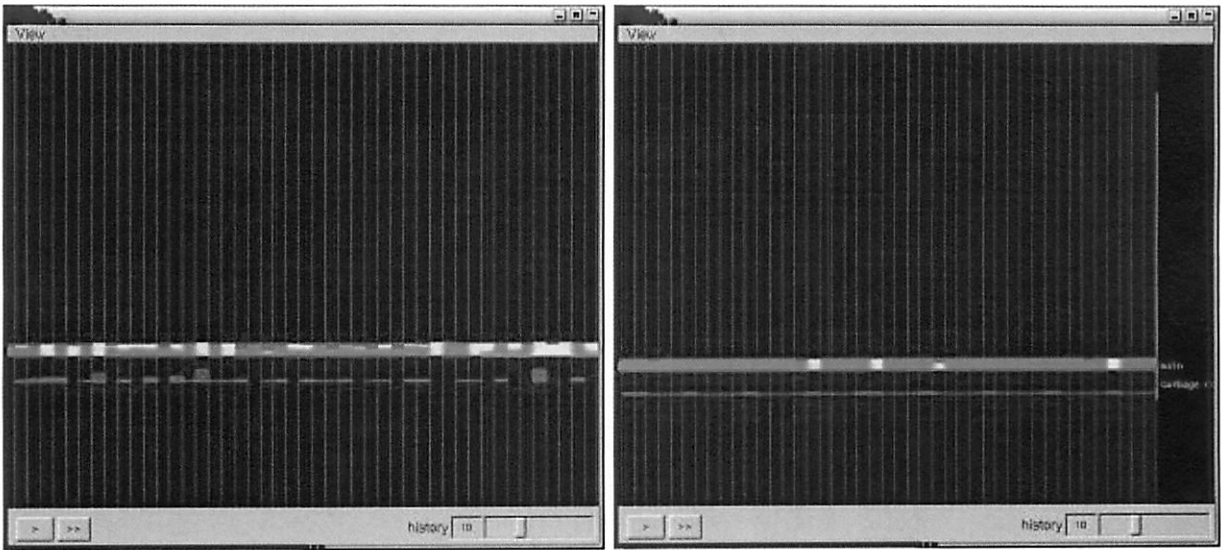


Figure 5: The garbage collector is unusually active in the left image, compared to proper execution on the right.

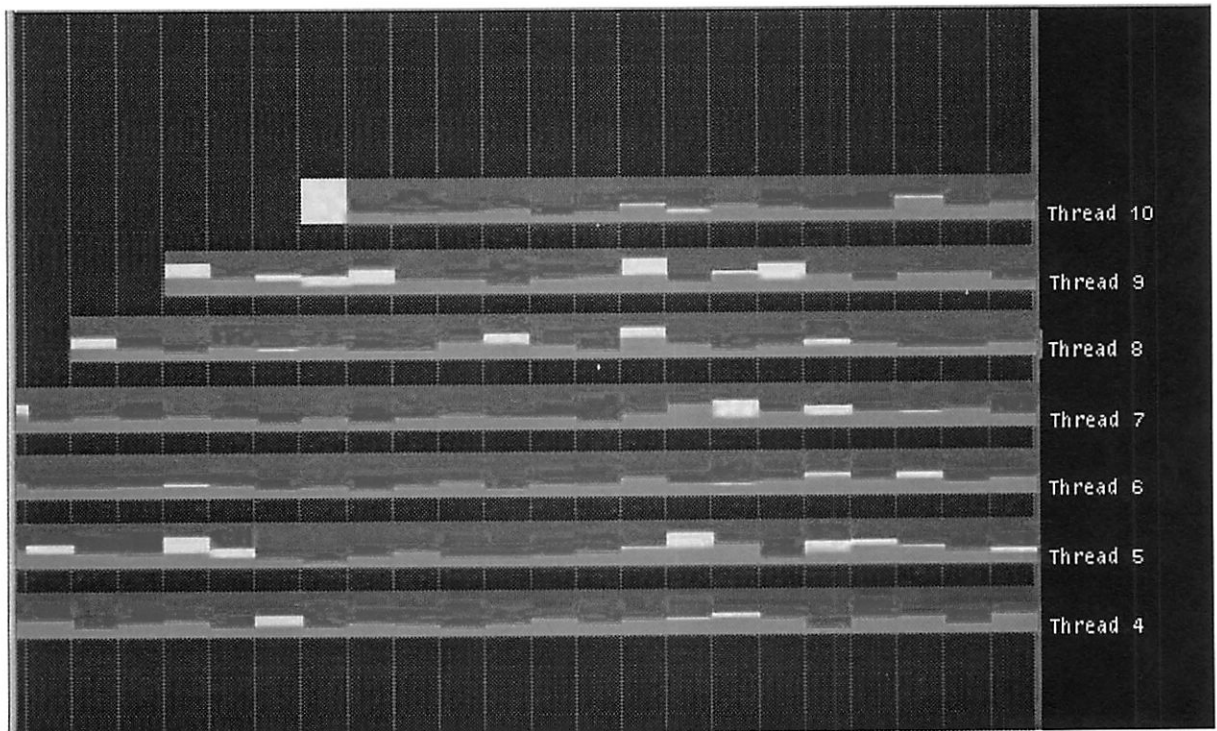


Figure 6: All of the threads running approximately the same amount of time illustrates that `setPriority` isn't working.

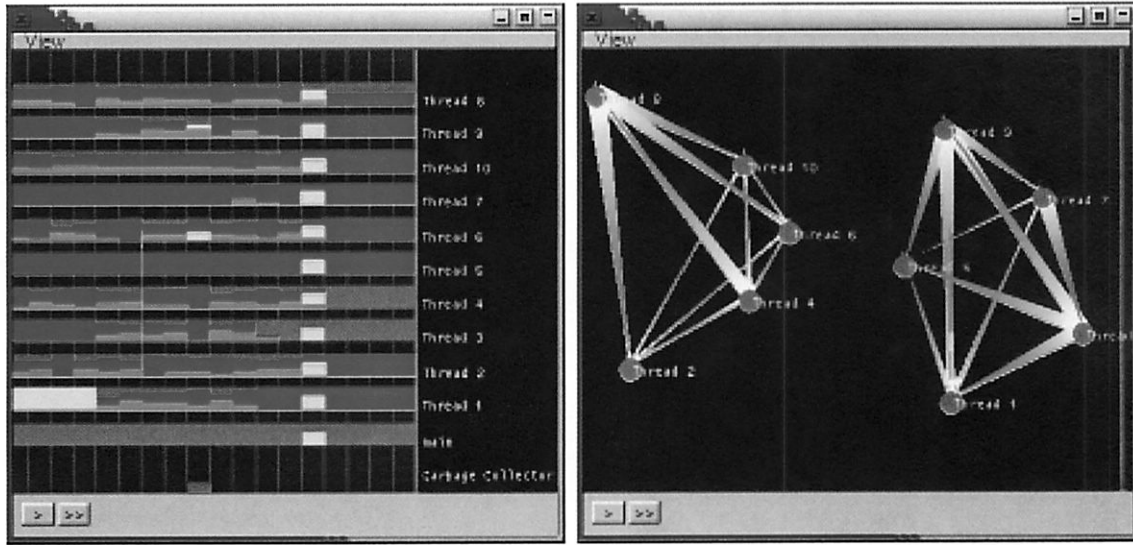


Figure 7: In the timeline view it appears that all threads are sharing the same lock. The interaction view lets us see there are actually two separate groups of threads interacting.

in the runnable state. This is not what was expected. The culprit turned out to be that Java does not pass priority information up to the LWPs, thus calls to `setPriority` have no effect when the underlying thread library is POSIX based (this is not true of the Windows NT thread library).

Unless the programmer happens to know this, they will often set priorities and have no idea that they are not having any effect. This can lead to programs not behaving as expected and experiencing decreased performance. With TIV, it is possible to examine thread scheduling information and identify any erroneous behavior.

6.3 Complex grouping behavior

In this program, a group of ten threads are heavily blocking as they contend for shared resources. There are two separate `HashMap` objects which are causing the blocking, and each thread is either using one or the other.

Figure 7 shows the timeline and interaction views. From the timeline view alone, determining which threads are sharing the same hash map is impossible. However the interaction view clearly shows which threads are blocking on each other, making this an easy task.

6.4 Deadlock detection

It is typically very difficult to identify a deadlock situation, as the resulting behavior is that the deadlocked threads simply stop responding. Using the two views of TIV, deadlocks are easy to identify. Figure 8 shows a deadlock. The timeline view identifies the deadlocked threads by them being completely in the blocking state. This is similar to the way other tools identify deadlocks. However the interaction view makes these even easier to identify. In that view, we can see that they are blocking on each other. This alleviates any shadow of a doubt that they are deadlocked, and not individually doing a lot of blocking.

6.5 Message Server model

This program is a socket based message server. We wrote it to simulate a system being used at Oculus Technologies Corporation. It demonstrates a situation where the distinction as to why a thread is blocking is

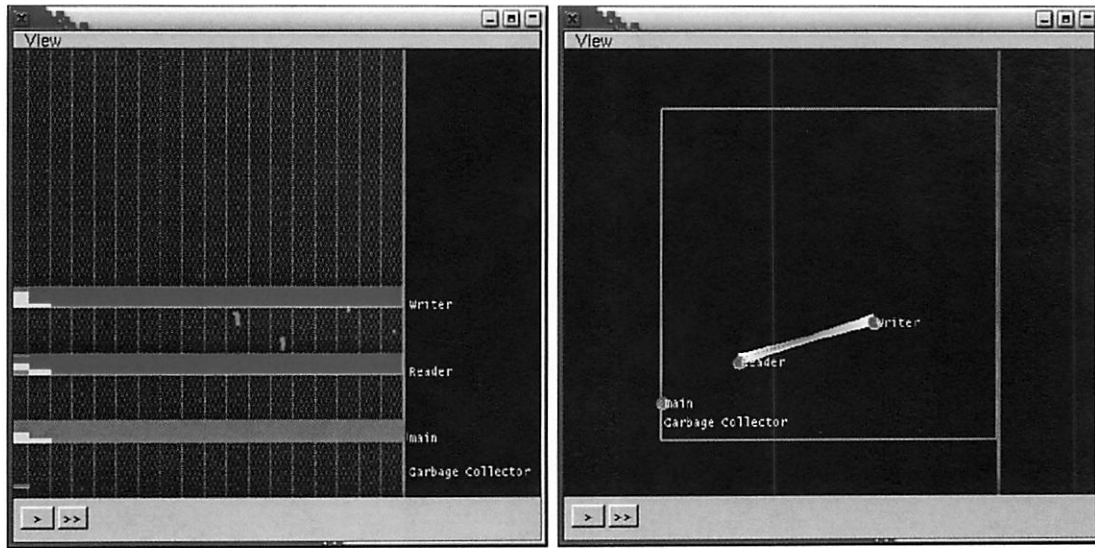


Figure 8: In the timeline view it appears that all threads are sharing the same lock. The interaction view lets us see there are actually two separate groups of threads interacting.

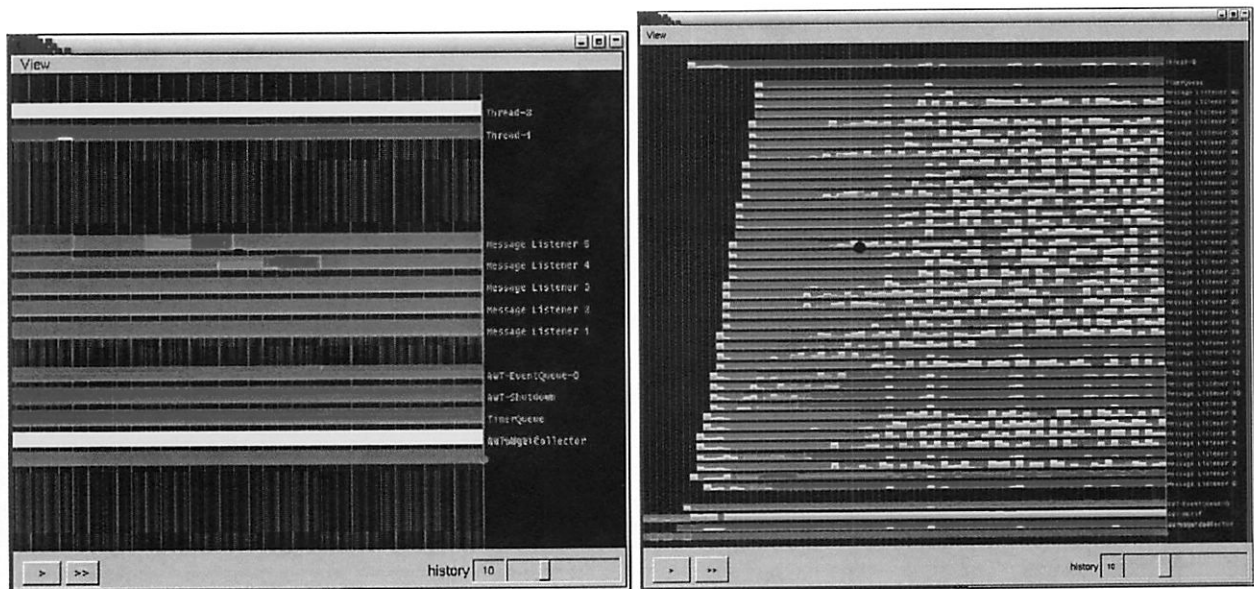


Figure 9: In this socket server, blocking on a socket and blocking on a mutex can be distinguished because of TIV's inclusion of the *I/O waiting* state.

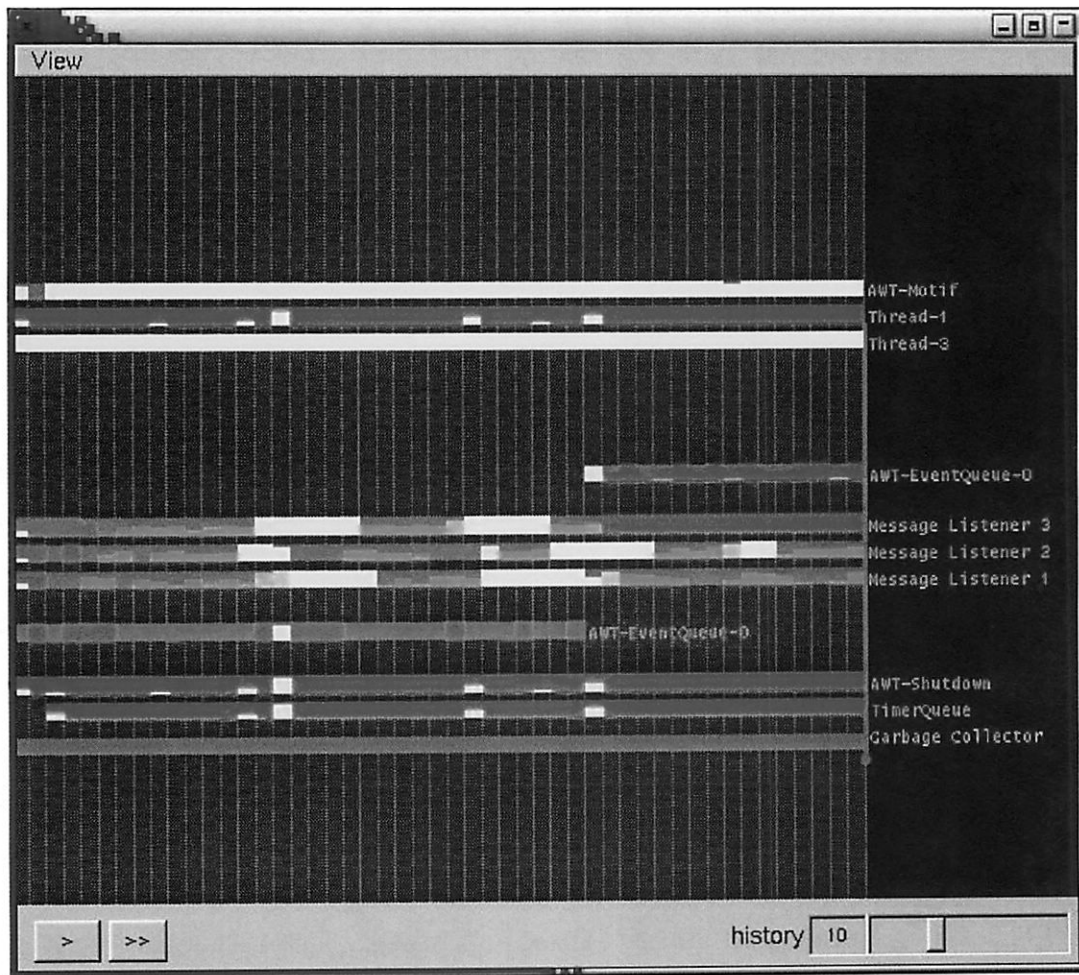


Figure 10: In this socket server, blocking on a socket and blocking on a mutex can be distinguished because of TIV's inclusion of the *I/O waiting* state.

important, and shows how TIV can be used to identify problems that occur from complex thread interactions. It also demonstrates TIV's effectiveness on programs with large numbers of threads.

This server maintains a pool of "listener" threads, each of which waits on a socket for incoming messages. When a message arrives, one of the listeners receives it, handles it, and updates the display. All updates take place serially on the "AWT Event" thread through a call to `java.SwingUtilities.invokeLaterAndWait` (in Java, this is the only way ensure proper behavior when updating Swing components). The update takes a few seconds to finish. Since the listener relies on the output for its execution, it blocks until the update is done. When a listener is not handling a message, it blocks on the mutex its socket holds. This blocking constitutes correct behavior – when a thread is blocking that means it is available to receive a message. Incorrect blocking also sometimes occurs when the listeners are handling messages.

This is where a problem comes in: these two types of blocking need to be distinguishable. To debug programs of this nature, it is necessary to be able to tell if blocking is supposed to be happening, or if it's a problem. Without doing so, any time a thread is blocking it will simply appear as if it were ready to receive a message. TIV makes this distinction possible through the inclusion of the *I/O waiting* state, and to the best of our knowledge is the only tool to do so. In figure 9, different colors differentiate between I/O and other blocking. The figure on the left represents a correctly executing program, with the five threads clustered in the middle the listeners. Two of them receive a message, on which they spend a few seconds executing (green) then a few second blocking (blue). The blocking is immediately distinguishable from the socket blocking with TIV.

The figure on the right shows a more complex server with forty listeners in which a problem occurs. In the first few seconds no messages are comgin in, so all the listeners are waiting on I/O (red). Then messages start coming in, and the listeners become active. After a few seconds, a group of threads all get stuck blocking on a lock as one thread refuses to give up control. These threads are easily identifiable with TIV as they can be identified as being in the blocking state. With other tools, it would be difficult to tell if they were acting erroneously, or simply ready to receive a message.

Another problem that TIV helps identify can be seen in figure 10. In this example, a problem occurs when a listener sends a request to update the display on the AWT Event thread. As mentioned previously, the listener suspends itself while the AWT thread is doing the update. The listener relies on the AWT thread to wake it back up when the update is complete. However, in this case the AWT thread hits an `ArrayOutOfBoundsException` and dies before it has a chance to do so. Java's Swing library is relatively intelligent, so the instant the AWT Event thread dies a new one is spawned to take its place. This is convenient, as it means the entire program does not crash when a single thread makes a mistake. However it also serves to mask a problem. In this case, the listener that was waiting on the AWT thread will remain suspended forever. As the program remedied the AWT thread crashing problem quietly, this will be the only behavior that is noticeable. Determining that another thread was the cause of the listener threads problem is not easy. However, TIV provides a way to identify this situation. In figure 10, the death of the original AWT Event thread and the spawning of the new one is depicted. It can also be seen that "Message Listener 3" stops responding at the same time. Identifying the problem after this correlation has been made is a much easier task.

6.6 Webcrawler

Dr. Reiss has a program that crawls the web and parses HTML files. It spawns 32 threads that each work independently parsing words. At one point he found that for some odd reason, the program was incredibly slow.

The problem turned out to be unexpected blocking. All of the threads were checking words against a dictionary stored in a java implemented hash table. Unbeknowst to him, the hash table synchronizes all calls, even reads. This resulted in a huge holdup as 32 threads fought over the lock on the hash table. Since all the threads were doing was reading from the table, no synchronization was necessary, and none was expected.

The way he discovered the problem was incredibly tedious and probably not possible for any other human being. He halted the program at random times in a debugger and analyzed the state of the program, eventually gathering enough evidence to determine that threads were spending a large amount of time blocking.

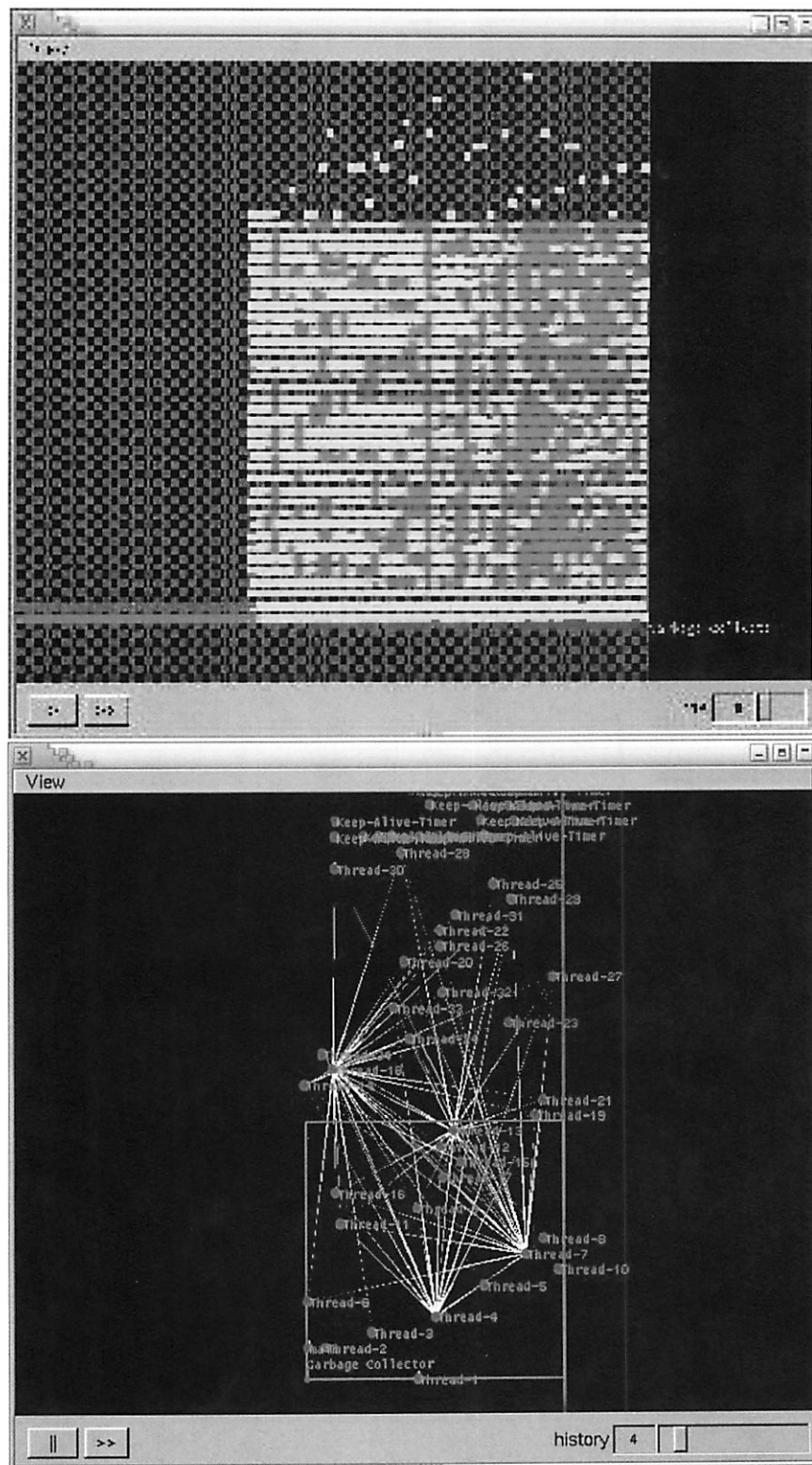


Figure 11: *Top*: Timeline view of the web crawler. *Bottom*: Interaction view shows highly connected but not especially significant blocking.

This type of problem is difficult to identify, as the data related to blocking is non-deterministic, difficult to output through log files, and hard to see if you are not looking for it. TIV, however, easily identifies problems like these.

Figure 11 shows the timeline and interaction views for a healthy run of this program. We were unable to obtain a trace of the broken code, but these pictures suffice to illustrate how TIV would identify the problem.

The interaction view shows that the crawler threads are virtually always in either the running or runnable states, with a small amount of socket waiting. These threads are running very efficiently, computing for the most part and waiting for data infrequently. If they were blocking excessively, it would be readily apparent in this view, as blue would be the predominant color of the threads. The interaction view would illustrate why. In the bottom figure, we see blocking occurring on four “hot spots”; threads that all other threads are blocking on. This picture is showing four seconds of history. If we were to look at just one second, there would be a single hot spot. In this example, a lock is being swapped between threads every second or so. However the blocking is very minimal and not a significant concern. In the case where there was a lot of blocking, this view would show a highly dense web of blocking. The threads would be fully connected (or at least close to it), which would indicate that they were all sharing the same lock.

7 Conclusion and future work

In this paper we have presented TIV, a visualization tool designed to analyze the behavior of threaded applications. It analyzes stored traces of Java programs, although can easily be extended to work with other languages and thread packages. TIV uses two primary views, which when used together offer an identification of different patterns of blocking behavior.

TIV allows a detailed analysis of the application and visualizes both thread scheduling activity and I/O activity. This combination provides the ability to distinguish between blocking that is supposed to occur (such as blocking on a socket) and blocking that isn't (such as excessive contention on a lock). We have shown how this proved invaluable in debugging a message server. We have also shown how TIV can help identify a number of other issues that can come up in threaded applications.

TIV provides a visualization that scales to large programs through the implementation of levels of detail. These levels allow TIV to maintain an uncluttered, readable visualization for programs of up to several hundred threads.

TIV increases upon the capabilities of previous tools, both in the ability to identify behavior and to examine large scale programs. This tool provides an excellent way of examining a threaded application and gaining an understanding of its behavior. It makes it easy to spot bad behavior, which can then be used to find the problem and fix it. This tool will be most useful when used in conjunction with a suite of tools, such as the DESERT environment.

In the future, we would like to increase the size of the programs TIV visualizes and make it even more accessible. We have used TIV to analyze programs ranging in size from 5 - 60 threads. As a next step, We would like to explore programs of up to 100 - 200 threads.

Additionally, we have found that TIV allows a fast enough examination of data that it could be used in close to real time. The next step is to eliminate traces and instead attach TIV to a running process. This would eliminate the slowest element of using TIV, taking the trace, and also allow it to be used on programs that run for any length of time. We will do this using the `tmon` interface, a close cousin of `tfile`, that outputs data from a running Java program. Its impact is minimal, meaning we could examine a program with a slowdown factor of around four or five.

8 Acknowledgements

I would like to thank the members of my committee, David Laidlaw, Steve Reiss, and Shriram Krishnamurthi, for the incredible amount of help they have given me in completing this project.

Special thanks to Pete, the whiz kid who provided me with much of the Java networking code I used in my sample programs. If you're a professor and you're in charge of admissions, check this kid out in a few

years.

9 Appendix A: TIV data gathering API

TIV defines an API to gather thread information. Any language or thread package that can satisfy the calls in this package can be used with TIV. This API is general enough that it should work for most standard thread implementations. There are eleven functions that have to be called.

Thread creation and deletion

- `signalThreadStart(thread id, time)`
- `signalThreadEnd(thread id, time)`

These functions report when a thread is spawned and when it dies.

I/O Routines

- `signalIOStart(thread id, type, time)`
- `signalIOEnd(thread id, type, time)`

These functions report when the program is entering and exiting I/O. `type` is either disk I/O or network I/O. In Java, the calls are: `read()`, `readByte()`, `connect()`, and `accept()`.

Monitor events

When threads enter the wait queue, enter, and exit a monitor

- `signalMonitorWait(thread id, monitor id, time)`
- `signalMonitorEnter(thread id, monitor id, time)`
- `signalMonitorExit(thread id, monitor id, time)`

`wait` means the thread has entered the wait queue for the lock.

`enter` means the thread has gained control of the lock.

`exit` means the thread has released the lock.

Suspension events

- `signalSuspendedStart(thread id, time)`
- `signalSuspendedEnd(thread id, time)`

These functions report when a thread has its execution suspended. In Java, we get this from the functions `wait()`, `sleep()`, `suspend()`, and `join()`.

Garbage collection information

- `signalGarbageCollectorEnd(time)`
- `signalGarbageCollectorStart(time)`

These functions report when the garbage collector is active. If the language does not implement garbage collection, these calls can be omitted.

References

- [1] Qiang A. Zhao and John 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.
- [2] John T. Stasko, The PARADE Environment for Visualizing Parallel Program Executions: A Progress Report. *Technical Report GIT-GVU-95-03, Graphics, Visualization, and Usability Center, Georgia Institute of Technology, Atlanta, GA*, 1995.
- [3] B. Stein and Chassin de Kergommeaux, Interactive Visualization Environment of Multi-threaded Parallel Programs.
- [4] J. Guitart, J. Torres, E. Ayguad and J. Labarta. Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications. *2nd Workshop on Java for High Performance Computing, Santa Fe, New Mexico (USA)*, pp. 15-25. May 2000.
- [5] <http://www.research.ibm.com/jinsight/docs/index.htm>
- [6] Michael Golm, Christian Wawersich, J. Baumann, M. Felser, and J. Kleinoder. Understanding the Performance of the Java Operating System JX using Visualization Techniques. *Workshop on Software Visualization, OOPSLA 2001, Tampa, FL*, October 15, 2001.
- [7] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D Editor: A New Interactive Parallel Programming Tool. *Proceedings of Supercomputing '94*, Washington, DC, November 1994.
- [8] Jack Dongarra, Orlie Brewer, James Arthur Kohl, and Samuel Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9(2):198-202 June 1990.
- [9] Wim De Pauw et al, Drive-by Analysis of Running Programs
- [10] John May and Francine Berman, Creating Views for Debugging Parallel Programs.
- [11] Bryan M. Cantrill and Thomas W. Doeppner Jr, ThreadMon: A Tool for Monitoring Multithreaded Program Performance.