TIV: A Thread Interaction Viewer

Kevin Audleman David H. Laidlaw Steven P. Reiss

{kforbes, dhl, spr}@cs.brown.edu
Department of Computer Science, Brown University, Providence, RI

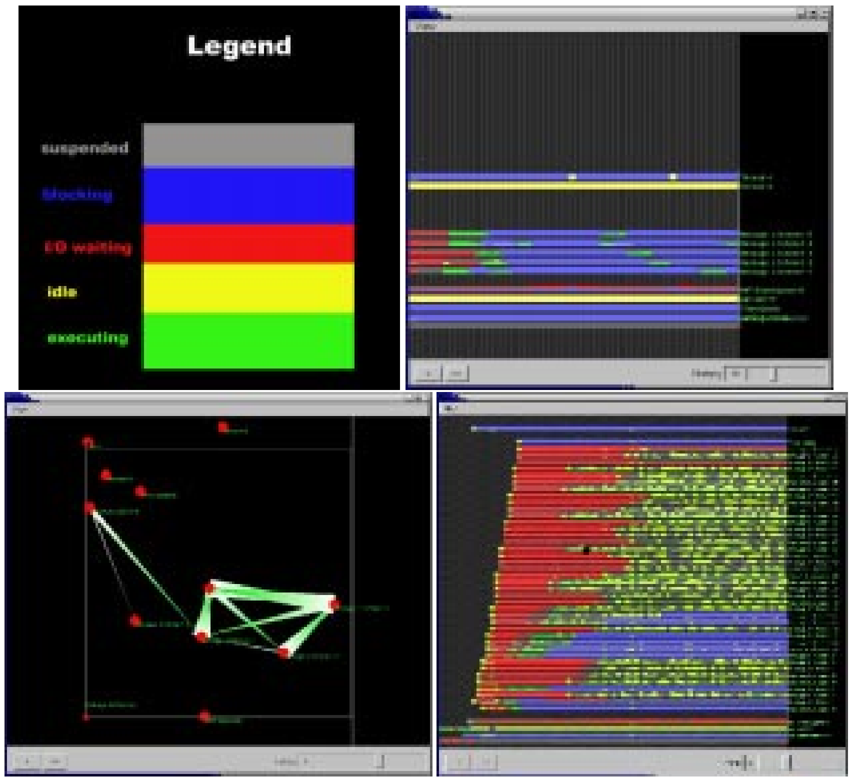

Figure 1: Clockwise, from top left: a) Thread states are represented as colored bars. b) The timeline view shows the states of execution of the threads, describing how they are acting. c) The interaction shows group behavior on mutual locks, describing how the threads are interacting. d) A larger view showed at the mid level of detail.

This paper presents Thread Interaction Viewer (TIV), a visualization tool for examining the thread behavior of large, long-running Java programs. It provides a novel view of thread scheduling information, I/O behavior, and blocking behavior in a manner that helps in the understanding and debugging of thread related problems. This is facilited in part by a flexible visualization framework that uses levels of detail to scales the visualization to best reflect the relevant information for different numbers of threads. TIV uses a 3-D space to represent the threads, but relies primarily on two specific views in this space: the Timeline View and the Interaction View. These views together paint a picture of behavior: the timeline view shows how each thread is acting, and the interaction view how they're interacting. We have found TIV to be good at identifying problems that are the result of threads interacting improperly, such as erroneous synchronization behavior, errors in thread scheduling, and more complex user defined thread communication errors.

Introduction.
Threaded programs have multiple threads of execution working in the same address space. They work in shared memory, communicating through the use of locking primitives. Properly coding the synchronization of these locks is necessary to prevent threads from interfering with each other, however doing so is a delicate and error prone process. Small flaws in logic can have a large impact on program performance. For instance, introducing a lock in the wrong place, such as on a heavily accessed class or object, can create a huge amount of excess blocking. The will result is a drop in performance as many threads wait on a lock. Identifying the cause of this sudden slowdown is not easy without the right tool.

TIV is designed to help the user identify thread communication based problems such as these. It uses a data visualization that uses two distinct views, the timeline view and the interaction view, to make problems of this type stand out. TIV's data representation allows for an efficient means of examining large and long running programs. We describe the details of the visualization below.

Visualization
TIV's data representation is a combination of thread scheduling

states, I/O behavior, and shared blocking statistics. For each thread, TIV reports the time spent in each of the following states:
Running: It is active on a processor
Runnable: It is ready to run, but there just aren't enough processors for it to get one.
I/O Waiting: It is currently waiting for a call to a memory or on a socket.
Sleeping: It is waiting for a synchronization variable.
Suspended: A call to the suspension function `thread.suspend()` has been made.

In the examples in this paper, each state is represented over a one second time window. However, this length of time is user controllable to allow for a coarse or fine examination of the data. An example will illustrate the use of TIV.

Example
Figure 1 illustrates a message server. It has five listener threads waiting on sockets to receive messages, which it then processes and displays in a GUI. For some reason, it is running slowly. As we will demonstrate below, TIV makes it easy to identify the problem.

Figures 1b shows the **timeline view**, which is used to examine states. The five threads in the middle are the listener threads. In a proper execution, we would expect these threads to be primarily in the I/O waiting state, ready to accept incoming messages. However the timeline view shows us that they're mostly in the sleeping state. This explains the slowdown – the listeners are not available to accept incoming messages because they're spending all of their time blocking. The interaction view gives more detail.

Figure 1c shows the **interaction view**, which is used to examine blocking behavior. A connection between two threads `T1` and `T2` indicate that one of them is waiting on a lock the other holds. In this example, it can be seen that all of the listeners are waiting on each other. This indicates that there is a lock they are all fighting over, which is unexpected. Examination of the code reveals that each thread is accessing a Java implemented `hashTable` that requires synchronous reads. Switching to a `hashMap` removes the synchronization and eliminates the slowdown.

Levels of Detail
TIV also gives a means for examining programs with a large number of threads through a flexible visualization framework. We have tested its effectiveness on programs up to 50 threads, and expect to increase this to about 200. LOD changes the visualization based on how many threads are on the screen. There are three levels: **near**, **mid**, and **far**. Near is as described above, and mid and far successively compact the information displayed. This serves the purpose of keeping the visualization readable at large scales – five states for 100 threads produces an incomprehensible display, but two or three colors is much more understandable. It also serves the purpose of abstracting away detail that is not as relevant as the number of threads increase. For instance, examining the scheduling of threads onto processors (the proportion of *running* to *runnable*) is useful with a small number of threads, as it provides a means of finding problems with priorities, such as threads being starved. However with twenty or more threads, it is often useful to assume the thread library is doing a decent job. Mid level handles this issue by compressing *running* to *runnable* into *executing*. The levels are describes as:

- **Near:** This is the same as described above. It describes the states: running, runnable, I/O waiting, sleeping, and blocking. It is useful for up to 20 threads.
- **Mid:** This level shows: executing, I/O waiting, sleeping, and blocking. It is useful for up to 50 threads.
- **Far:** This level combines *blocking* and *I/O Waiting* into the single category *Non-runnable* to further reduce visual clutter. This level is designed to point out potential bad behavior that can then be examined in greater detail with a closer view. The states it describes are: executing, non-runnable, suspended. It is designed for up to 200 threads.

Figure 1d gives an example of TIV running at mid level. This is a message server with forty message listeners. The listeners initially start in the waiting on I/O state, but then are deluged by a massive number of messages. This can be seen by the large amount of time they spend in the *executing* state. The specifics of when each thread transitions from runnable to running aren't of any concern with this program, thus the mid level of detail is the right level to look at this data.

Conclusion
TIV is a visualization tool that aids in the process of debugging and understanding large threaded programs. It uses two main views, the timeline view and the interaction view, to communicate how threads are acting and interacting. It utilizes a level of detail system to focus the visualization for different numbers of threads. TIV is useful for identifying problems that occur through improper thread communications, and is effective for programs with widely varying numbers of threads.

References

[1] Qiang A. Zhao and John T. Stasko, Visualizing the execution of threa ds-based parallel programs. Technical Report GIT-GVU-95/01, Graphics, Visualiza tion, and Usability Center, Georgia Institute of Technology, Atlanta, GA, Januar y 1995.

[2] John T. Stasko, The PARADE Environment for Visualizing Parallel Prog ram Executions: A Progress Report. Technical Report GIT-GVU-95-03, Graphics, Vis alization, and Usability Center, Georgia Institute of Technology, Atlanta, GA, 1 995.

[3] B. Stein and Chassin de Kergommeaux, Interactive Visualization Envir onment of Multi-threaded Parallel Programs.

[4] Jordi Guitart, et al, Java Instrumentation Suite: Accurate Analysis of Java Threaded Applications.

[5] http://www.research.ibm.com/jinsight/docs/index.htm

[6] Wim De Pauw et al, Drive-by Analysis of Running Programs

[7] John May and Francine Berman, Creating Views for Debugging Parallel Programs.

[8] Bryan M. Cantrill and Thomas W. Doeppner Jr, ThreadMon: A Tool for M onitoring Multithreaded Program Performance.